Performance Considerations 4

This chapter describes programming approaches to maximize performance, reports Intel740™ graphics accelerator performance test results, and introduces creative programming techniques which take advantage of the Intel740™ graphics accelerator chip features.

4.1 Performance Strategies And Measurements

All performance statistics outlined in this section were gathered using Intel's RasM (Raster Metric) 2.0 software. RasM, a raster speed measurement tool, measures the rasterization speed of a hardware accelerator vs. the scene complexity of an application. The system configuration used for gathering the data shown in this document is as follows:

- 300 MHz Pentium® II processor with MMX™ technology
- Atlanta motherboard with PhoenixBIOS*
- Intel® 440LX AGPset
- Intel740™ graphics accelerator AGP graphics card with 200 BIOS
- Windows95 operating system (OSR2.1)
- 64 Mbytes system memory (SDRAM, 66 MHz)
- 4 Mbyte local video memory (SDRAM, 100 MHz)
- 640x480x16 bits per pixel screen resolution
- D3D test use execute buffers and OpenGL tests use vertex buffers with glDrawArray unless otherwise specified
- 60 Hz refresh rate

4.1.1 Intel740™ Graphics Accelerator Performance Capabilities

The Intel740™ graphics accelerator supports the next generation of high-content applications. 3D games will use more realistic models with more triangles of smaller size. The Intel740™ graphics accelerator provides its peak performance for these types of games.

The recommended game detail target for the Intel740™ graphics accelerator is 10,000 triangles per frame, between 75 and 175 pixels per triangle, at 30 frames per second. 10,000 triangles per scene requires a triangle rate of approximately 300,000. The Intel740™ graphics accelerator can render 366,000 full featured triangles per second with an average of 105 pixels per triangle.

```
Required_Tri_Per_Sec =Tri_Per_Scene / (1/Frames_per_Second - Tover_head)
```

The following sections include Intel740™ graphics accelerator performance results along with descriptions of how the results can be used to predict frame rates for particular applications and scene complexities.

4.1.2 Using CPU/Intel740™ Graphics Accelerator Concurrency

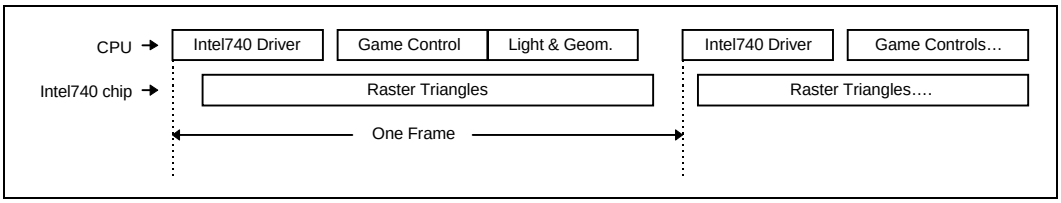
Applications should be designed to take advantage of the concurrency allowed by the Intel740™ graphics accelerator and AGP system architecture. The Intel740™ graphics accelerator can be thought of as a second processor for rasterization, optimized for maximum parallelism with the CPU. The benefit given to the application is that the CPU is free to do more AI, physics, lighting, and geometry. The Intel740™ graphics accelerator drivers minimize CPU overhead, balance the system, and allow for maximum system concurrency.

Many of the performance results included in this chapter report the driver duty cycle for the CPU. The duty cycle is the ratio of CPU time used by the Intel740™ graphics accelerator driver divided by the length of time the Intel740™ graphics accelerator requires to render the scene. It is a measure of how much times an application can spend on lighting, geometry, and game controls while not causing the CPU to limit performance.

In systems with software rasterization only, a typical application used 90% of CPU cycles for rasterization alone. Because the Intel740™ graphics accelerator renders much faster than software engines and because of the system's available concurrency, a system with an Intel740™ graphics accelerator gains a tremendous performance advantage.

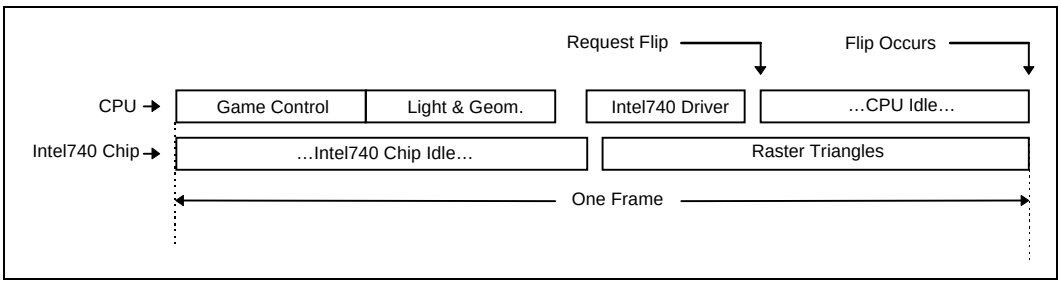
Figure 4-1 shows the usage model for the Intel740™ graphics accelerator and the CPU during one and a half frames of a typical application cycle.

Figure 4-1. Intel740™ Graphics Accelerator/CPU Usage Model



Applications should be structured such that CPU cycles are not wasted waiting for synchronization with the Intel740™ graphics accelerator. Forcing flips or blits to surfaces being rendered cause the CPU to sit idle until rendering has completed. Figure 4-2 illustrates how an improperly placed flip or blit can drastically reduce frame rate.

Figure 4-2. Improper Usage Model



In this case, only minimal concurrency is achieved. The problem can be alleviated by simply rearranging the flow so that the CPU processing for the following frame is completed before the render target is flipped. The problem can also be alleviated by triple-buffering. Similar problems will be seen by code that issues blit commands for 2D effects directly after sending a 3D scene.

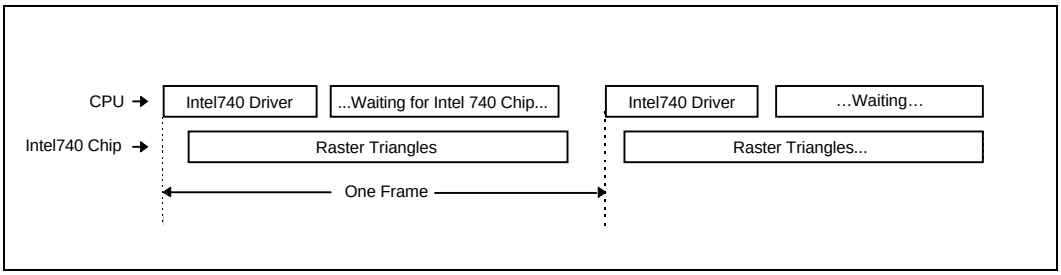
4.1.3 Performance Test Results

4.1.3.1 Raster Speed Test Method

This section describes the tests used to measure the performance numbers reported in this document.

Figure 4-3 shows the system usage while RasM is running. The time that RasM waits for the Intel740™ graphics accelerator to complete will be used for AI, game control, lighting, geometry, and anything else the application needs to do before sending the next frame to be rendered. To attain the maximum frame rate, applications should be optimized to finish all computations during this time.

Figure 4-3. RasM Intel740™ Graphics Accelerator/CPU Usage Model



The program execution can be divided into two phases called consecutively by a loop that sequences through all the triangle sizes to be tested:

```
Loop (for all triangle sizes do)
        Phase 1: Build buffers (execute buffer, vertex arrays, etc.)
        Phase 2: Execute the buffers and time the hardware
```

The first phase creates and fills execute buffers with 512 triangles each. The total number of triangles depends on triangle size, depth complexity (DC) goal, and percent Z-buffering (%Z) goal. Unless otherwise stated, the sweeps reported in this document have a constant DC of 2.5 and 50% Z across the triangle size sweeps. For example, the 120 pixel/triangle data point contains about 13,300 randomly distributed triangles per scene:

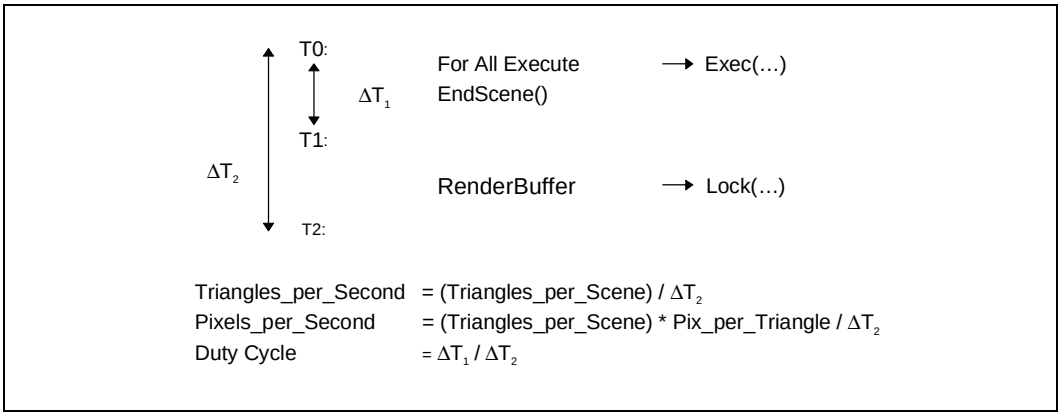
```
Triangles_per_Scene = (Screen.W * Screen.H * Avg_DC_Goal / Percent_Z_Goal) / Pix_per_Tri
```

To achieve a predefined DC and %Z goal, a “survival of the fittest” algorithm is implemented: 10 randomly placed and oriented triangles are generated for each required triangle, and the triangle that brings the scene the closest to its DC and %Z goals is selected.

Game scenes often have some percentage of triangles use specular and alpha blend. Many of the sweeps in this document test scenes with specular and blend enabled for only a fraction of the triangles. RasM always puts the triangles with specular and blend in the last portion of the scene. The rationale here is that games using only a small percent specular or alpha blend will be applying highlights to the scene near the end of their triangle lists. Unless otherwise stated, textures are mipmapped with 16-bit color. When multiple textures are used in a scene, they are distributed equally throughout the scene. A scene with 10,000 triangles, three textures, 30% specular, and 20% alpha blend would generate 20 execute buffers, 3,400 triangles per texture; the last 2,000 triangles would use specular and alpha blend, and the preceding 1,000 would use just specular.

The second phase of the loop executes the buffers created in the first phase, and then clocks the driver and hardware raster speed. The scene is clocked, displayed, and recorded 15 times; the middle five times are averaged to get the final result. Figure 4-4 shows pseudo-code from the timing/display loop.

Figure 4-4. RasM Pseudo-Code



The reported results are divided into sections: Result Summary, Basic Sweeps, Advanced Sweeps, and Full Sweeps. The Result Summary contains data taken directly from the set of sweeps. It is intended to be used as a summary or for quick reference. Section 4.1.3.2 contains more detailed information that can be used to predict application frame rates.

The basic sweep compares Gouraud only to Gouraud with Z-Buffer and, finally, Gouraud with Z-Buffer and Textures (GZT). The advanced sweeps takes the GZT features from the last basic sweep and tests the sensitivity to fog, alpha blending, specular, and anti-aliasing. The full sweeps combine all features.

Table 4-1. Result Summary

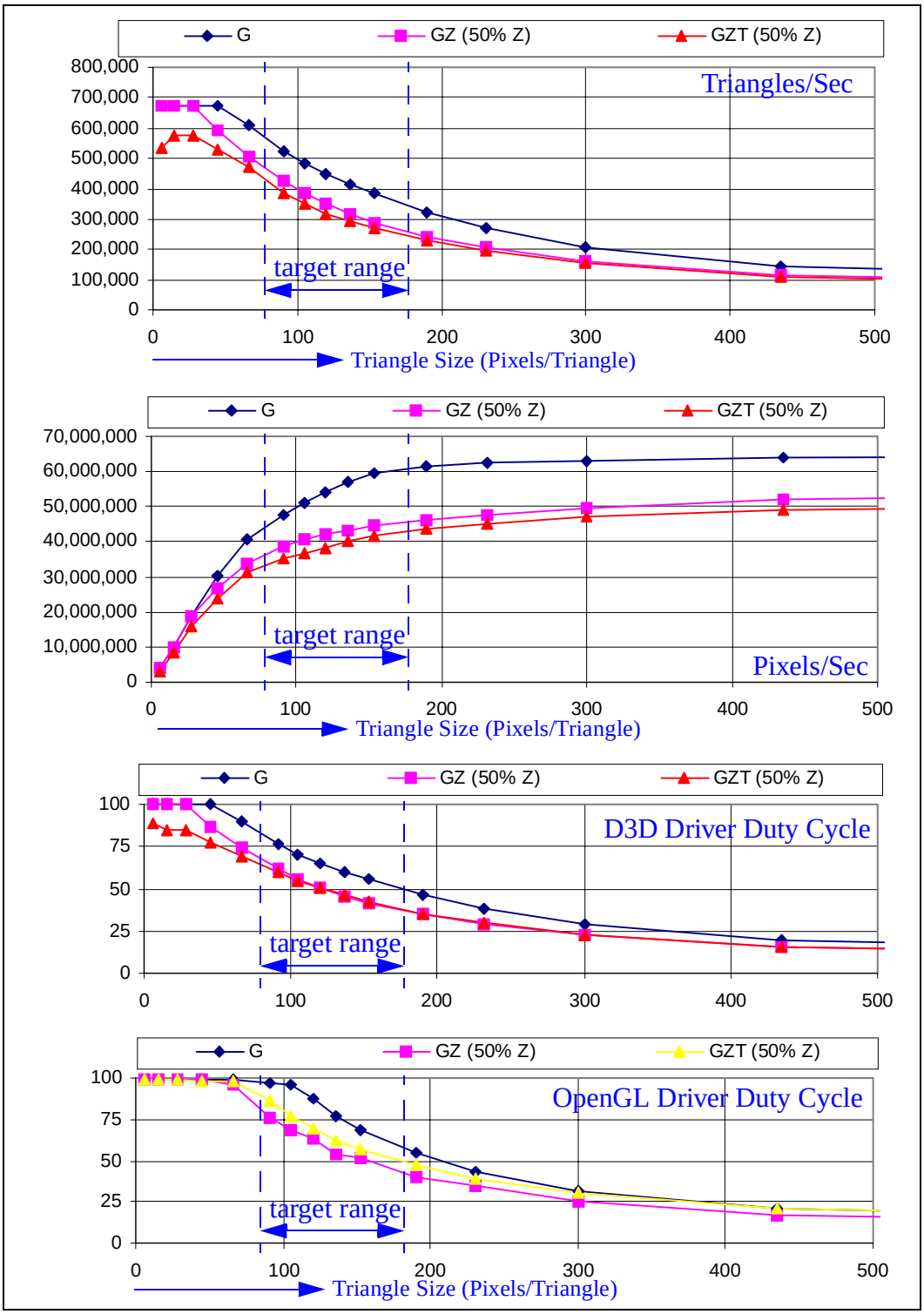
Gouraud	**Fog**	**Blend**	**Spec**	**Anti-Alias**	**Z-Buff 50% Z**	**MipMap 16 BBP**	**Set-up Limited** (Triangles per Second)	**105 PixTri** (Triangles per Second)	**Fill Rate** (Pixels per Second)
X							675k	482k	65.7M
X					X		672k	385k	59.1M
X					X	X	577k	349k	54.2M
X	X				X	X	535k	349k	53.8M
X		20%			X	X	568k	340k	53.8M
X		100%			X	X	535k	300k	48.2M
X			30%		X	X	539k	348k	53.7M
X			100%		X	X	468k	347k	54.2M
X				20%	X	X	372k	253k	51.4M
X	X	20%	30%		X	X	496k	338k	53.2M
X	X	100%	100%		X	X	415k	298k	48.0M

Table 4-2. Symbol Key

Symbol	Definition
G	Gouraud Shading
Fg	Vertex Fog Enabled 100% of Scene
A20 and A100	Alpha Blend Enabled for 20% and 100% of Scene
S30 and S100	Specular Highlights Enabled for 30% and 100% of Scene
Aa20	Anti-Aliasing enabled for 20% of Scene
Z	Z-Buffer enabled. Cleared at beginning of Scene
T	256X256 MipMap BiLinear Filter, ARGB 0565 Format (unless otherwise stated)

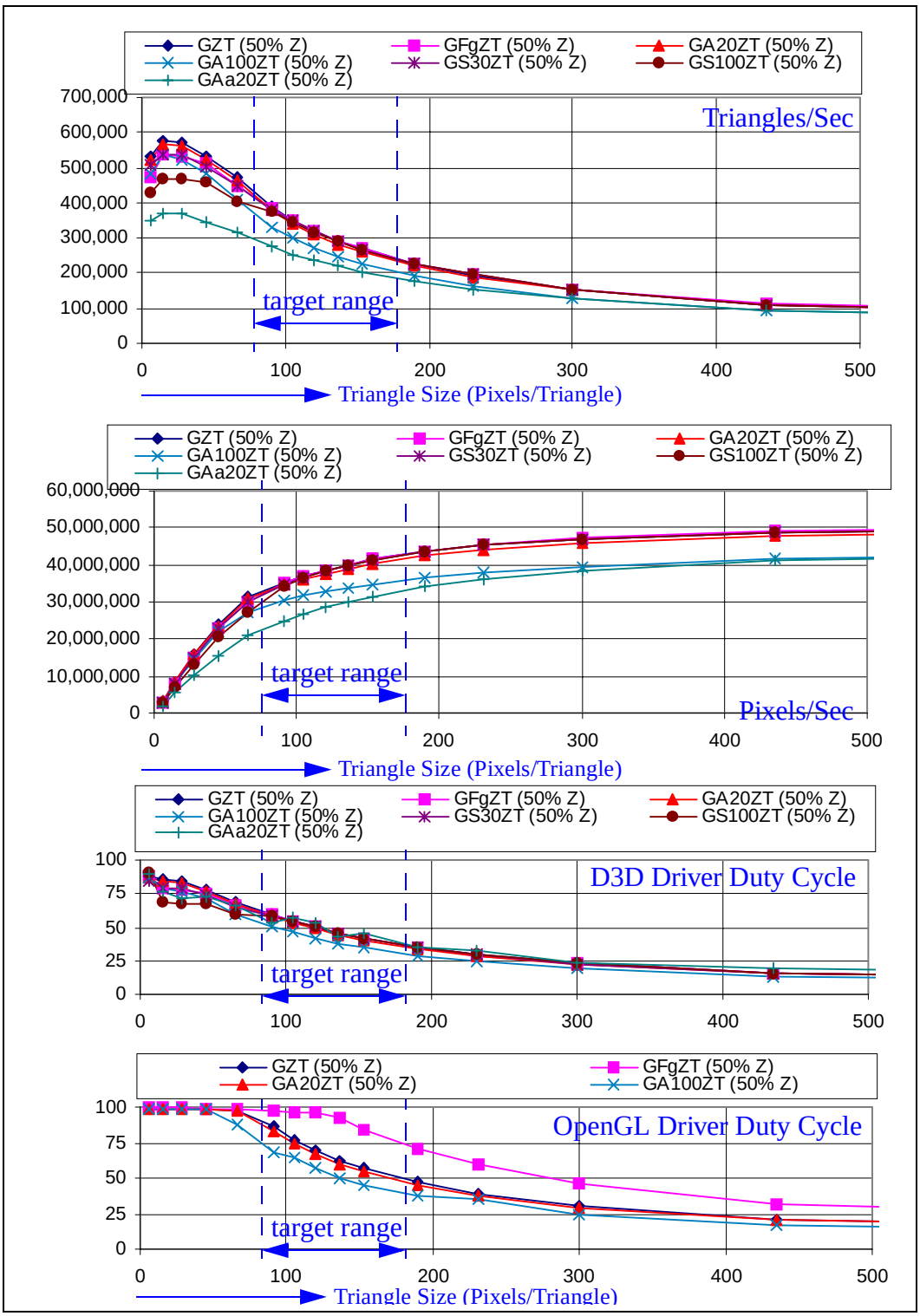
The graphs in Figure 4-5 show triangles per second, pixels per second, and duty cycle for Gouraud only, Gouraud with Z-Buffer and Gouraud with Z-Buffer and Textures.

Figure 4-5. Basic Feature Sweeps



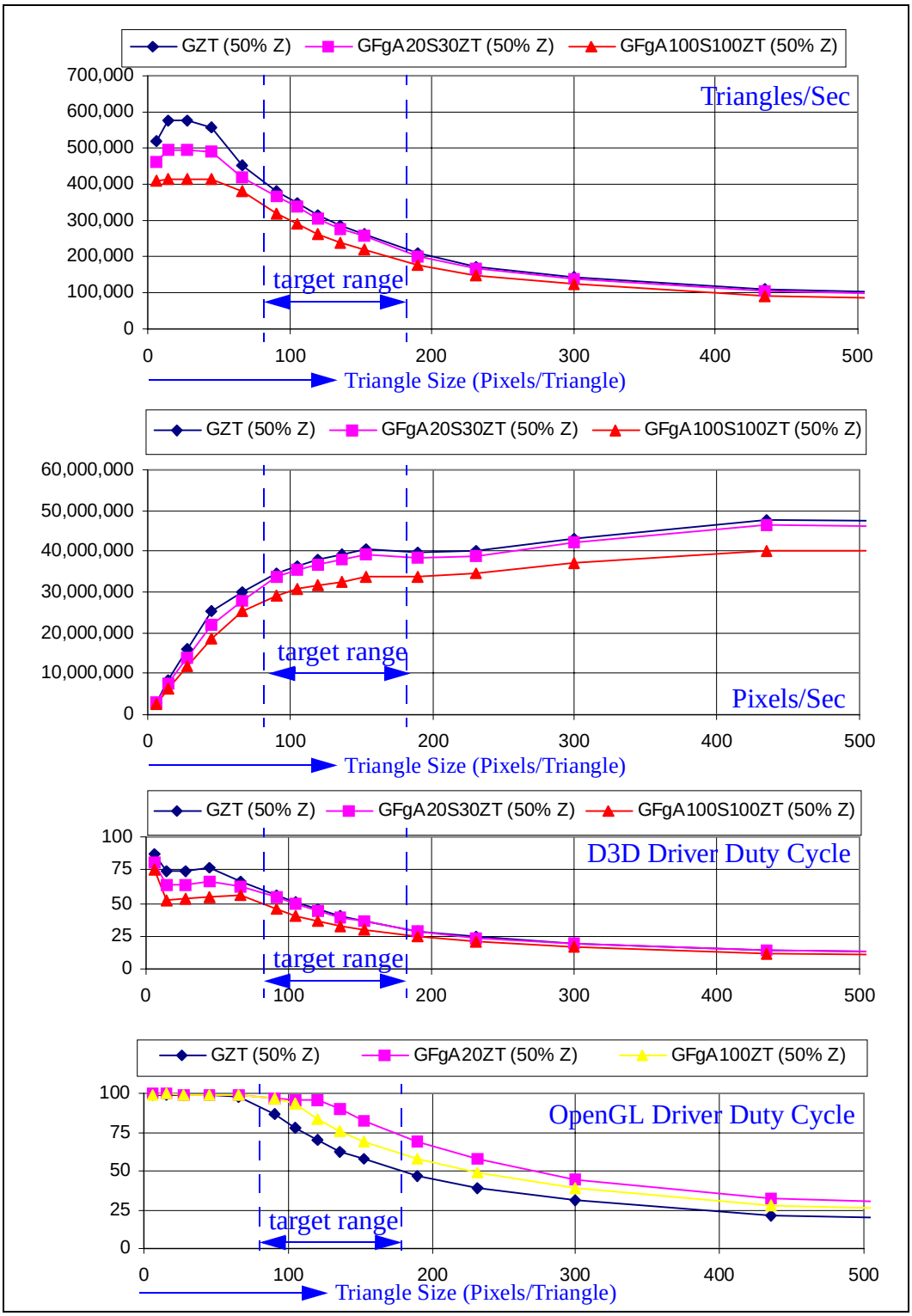
The graphs in Figure 4-6 show triangles per second, pixels per second, and duty cycle. The feature sets start with the GZT features set from the last basic sweep and display the sensitivity to fog, alpha blending, specular, and anti-aliasing.

Figure 4-6. Advanced Feature Sweeps



The graphs in Figure 4-7 show triangles per second, pixels per second, and duty cycle with full feature sets.

Figure 4-7. Full Feature Sweeps



4.1.3.2 Implications and Analysis

This section suggests how the reported results can be translated into performance for individual applications. The tests are raster speed only. Because of system concurrency, if the application code executed between scenes preserves the duty cycle, the stated triangle and fill rates will be achieved.

Average and percent Z are a good measure of scene complexity from the graphics card's point of view. They actually define the number of pixels that will be processed by the graphics accelerator, per scene. Pixels per scene and desired frames per second give the fill rate that is required of the graphics accelerator to hit that frame rate.

```
Pixels_per_Scene = (Screen.W * Screen.H * Avg_DC_Goal / Percent_Z_Goal)

Required_Fill_Rate = Pixels_per_Scene / (1/Frames_per_Second - Tover_head)
```

Tover_head is the overhead time which may be required to clear the Z-buffer, render buffers, or blit a background. The number and size of triangles per scene may be more convenient for a game designer to work with, but it is not a difficult conversion between the two.

```
Avg_DC_Goal = Triangles_per_Scene * Pix_per_Tri * Percent_Z_Goal / (Screen.W * Screen.H)
```

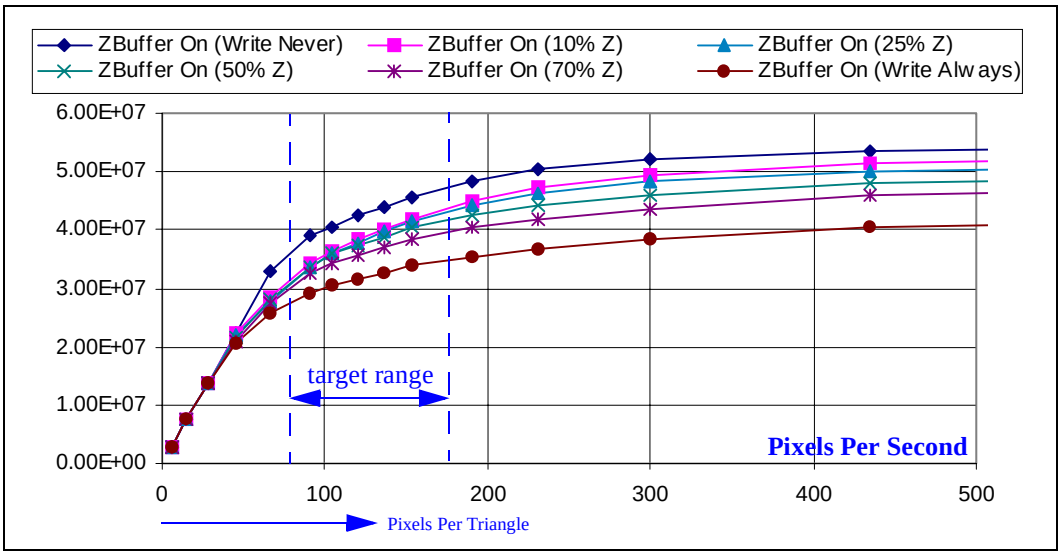
The %Z goal indicates how well the triangles are ordered before being sent to the Intel740™ graphics accelerator graphics accelerator. 50% Z assumes that half of the pixels contained in the processed triangles will actually not be written to the screen because they are behind the previous pixel in the z-order. Note that for a constant number of pixels per scene, if %Z goes up (a higher number of Z-values are written) then the DC also goes up. Even though the pixels per scene remains the same, the fill rate will change because it is a function of %Z.

A scene complexity of 2.5 DC and 50% Z was chosen because it is predicted that typical games will have a similar complexity. However, not all games will follow this pattern.

Depth complexity is a measure of pixels per scene. Increasing DC does not affect the triangle rate or the actual fill rate, but will affect the pixels per scene and the required fill rate according to the equations mentioned above.

Percent Z occlusion does affect the triangle and fill rates. Basically, decreasing %Z increases fill rate, and vice versa. Sorting triangles from front to back produces higher graphic card performance. Implementing a sorting algorithm is only recommended when the Intel740™ graphics accelerator fill rate becomes the system performance bottleneck. The following graph illustrates the performance with changing scene %Z occlusion.

Figure 4-8. Performance vs. Percent Z Occlusion



Very few games will have just one triangle size per scene, but it is useful to analyze just one size at a time because it supplies many of the building blocks required to approximate triangle rate, fill rate, and duty cycle for more complex scenes. This example uses a game scene of 7,000 triangles of 75 pixels and 3,000 triangles of 175 pixels, has a 50% Z, uses a full feature set of GFgA20S30ZT, has a Tover_head of about 1 ms, and requires 30 frames per second. The average DC for the scene comes to 1.71, the pixels per scene is 1.05M, and it requires a fill rate of 32.5M pixels per second.

Avg_DC_Goal = (75 * 7,000 + 175 * 3,000) * .5 / (640 * 480) = 1.71

Required_Fill_Rate = 1.05M / (1/30 - .001) = 32.5M

The fill rate for this type of scene is not explicitly quoted in the graphs included in this document, but a weighted average based on numbers of pixels can be used to extrapolate the Intel740™ graphics accelerator resultant fill rate. For the previous example, the extrapolated fill rate of the Intel740™ graphics accelerator is 35.2M pixels/s.

Pixels_per_Second (estimate) = (525k * 30.4M + 525k * 40.0M) / 1.2M = 35.2M Pix/s

RasM can be used to test scenes with non-constant triangle sizes. When the hardware was tested for this case, the actual fill rate was reported to be 34.2M pixels/s. Most of the discrepancy can be attributed to the scene depth complexity in this example being below that of the quoted tests. For more information on how DC (or total packet size) can affect performance, see Table 4.1.4.3 “Triangle Packet Size” on page 4-13.

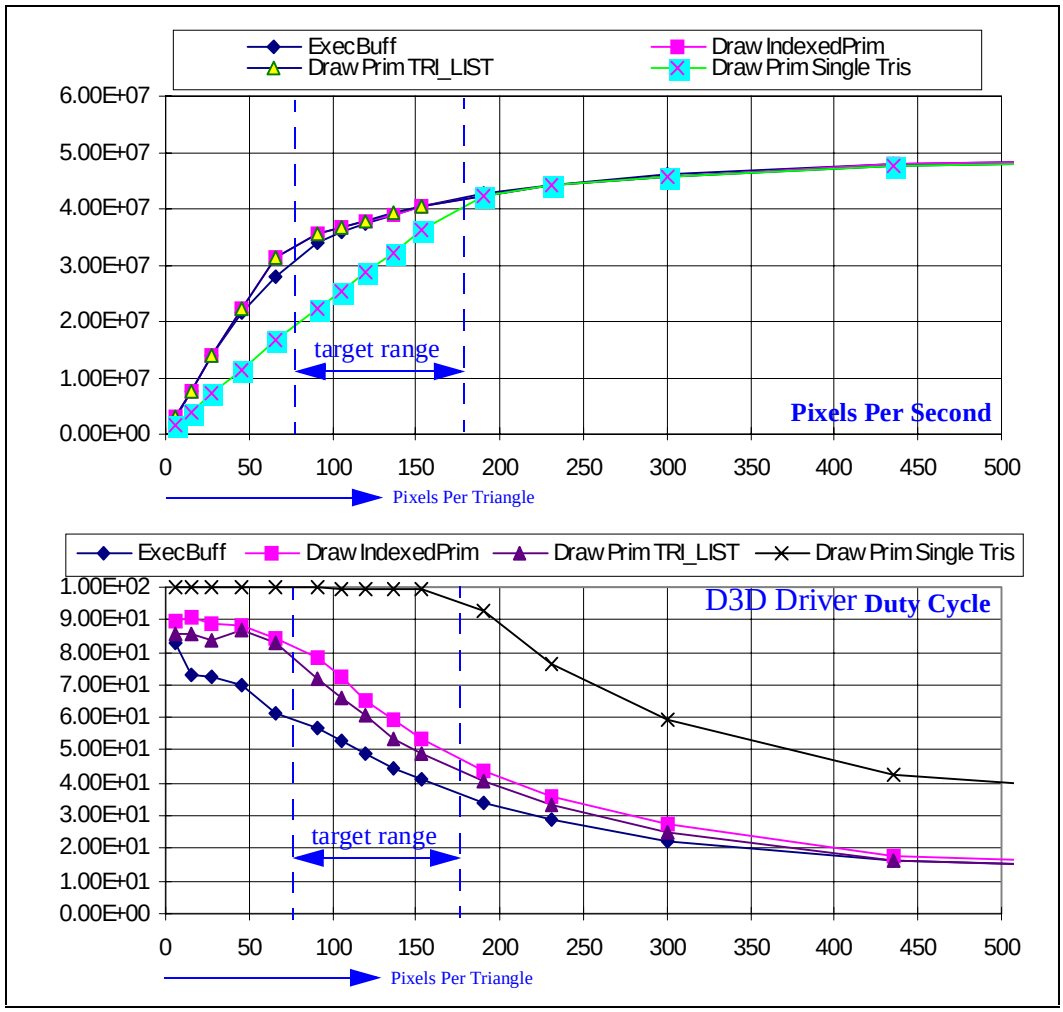
4.1.4 Special Performance Considerations

This section contains descriptions of subtle application design choices which can have considerable effects on performance.

4.1.4.1 Direct3D DrawPrimitive vs. Execute Buffers

Direct3D immediate mode allows programmers to choose between execute buffers and draw primitive methods of sending commands to the graphics hardware. The Intel740™ graphics accelerator performance and CPU driver duty cycle are both nearly identical for either sets of methods. This is the case as long as other considerations such as concurrency and packet size are not ignored. The following full feature sweeps (Fog, 20% Alpha, 30% Specular, MipMap Textures, 2.5 DC, 50% Z) use execute buffers, draw indexed primitive, draw primitive with triangle lists, and draw primitive with discrete triangles. Each of the instructions sending groups of triangles (includes all but draw primitive with discrete triangles) issues 500 triangles per instruction.

Figure 4-9. Performance of DrawPrimitive vs. Execute Buffer



Each method has an associated CPU overhead. Execute buffers have the lowest, followed by draw primitive with triangle lists. Sending a single triangle with each draw primitive command has a very high overhead; below about 200 pixels per triangle the CPU is unable to send enough triangles down per second to keep the Intel740™ graphics accelerator busy.

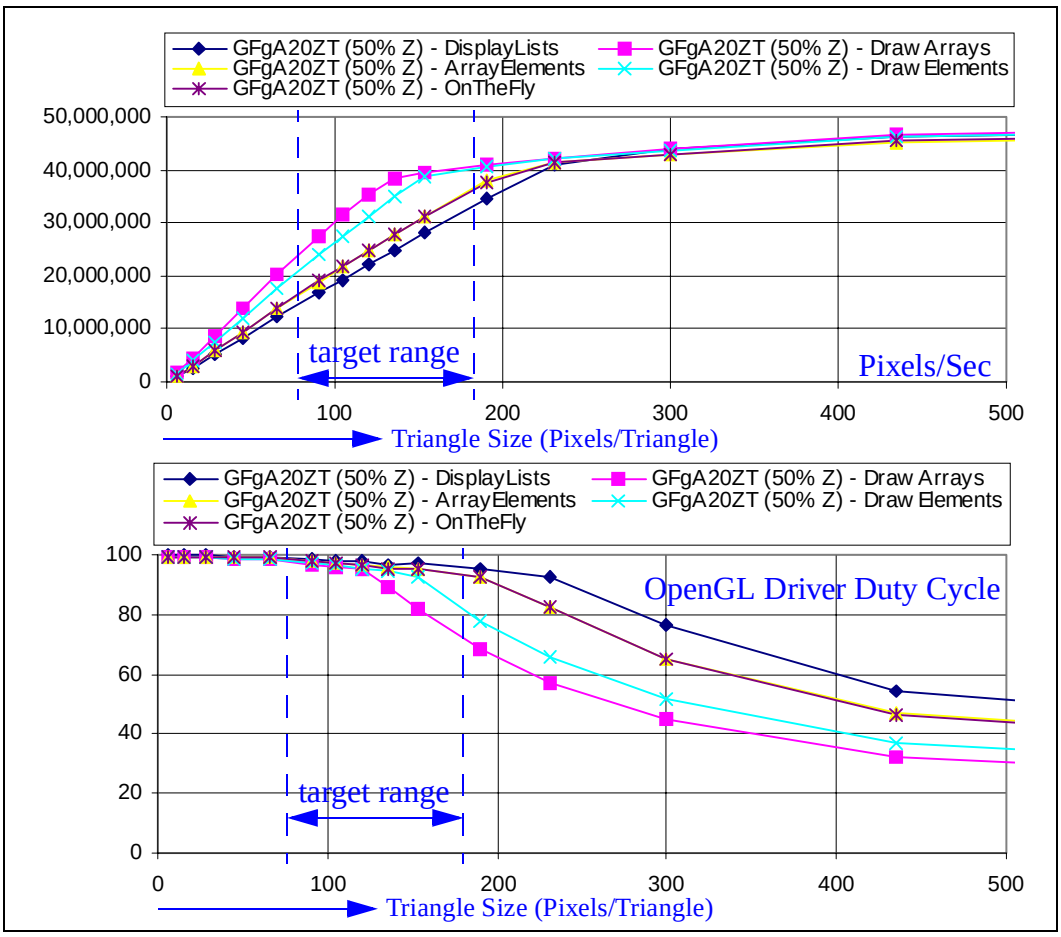
It is important to note that execute buffers tend to force applications to group triangle execution commands, which is advantageous for the Intel740™ graphics accelerator and its driver. For more information on Performance vs. triangle packet size see Section 4.1.4.3, “Triangle Packet Size” on page 4-13.

4.1.4.2 OpenGL Display Lists vs. Vertex Buffers

OpenGL give programmers the choice of several rendering methods: display lists, vertex buffers (using glDrawArray, glArrayElements, and glDrawElements), or simply specifying polygons and vertices on the fly. Vertex buffers are generally considered the highest performance method. Among the vertex buffer methods, glDrawArray and glArrayElement are the recommended methods for programming to the Intel740 graphic accelerator. Unless otherwise specified, all of the OpenGL driver duty cycle numbers in this manual use vertex arrays with glDrawArray.

The following full feature sweeps (Fog, 20% Alpha, MipMap Textures, 2.5 DC, 50% Z) use the various polygon rendering methods. The vertex buffer instructions use buffer of approximately 500 triangles.

Figure 4-10. Performance of Display Lists vs. Vertex Buffers



Each method has an associated CPU overhead. Vertex arrays with glDrawArray and glDrawElements have the lowest overhead. Sending triangle strips or fans either on the fly, in a display list, or vertex array is also an efficient method of batching primitives. In general, the more triangles batched in a single call, the lower the overhead.

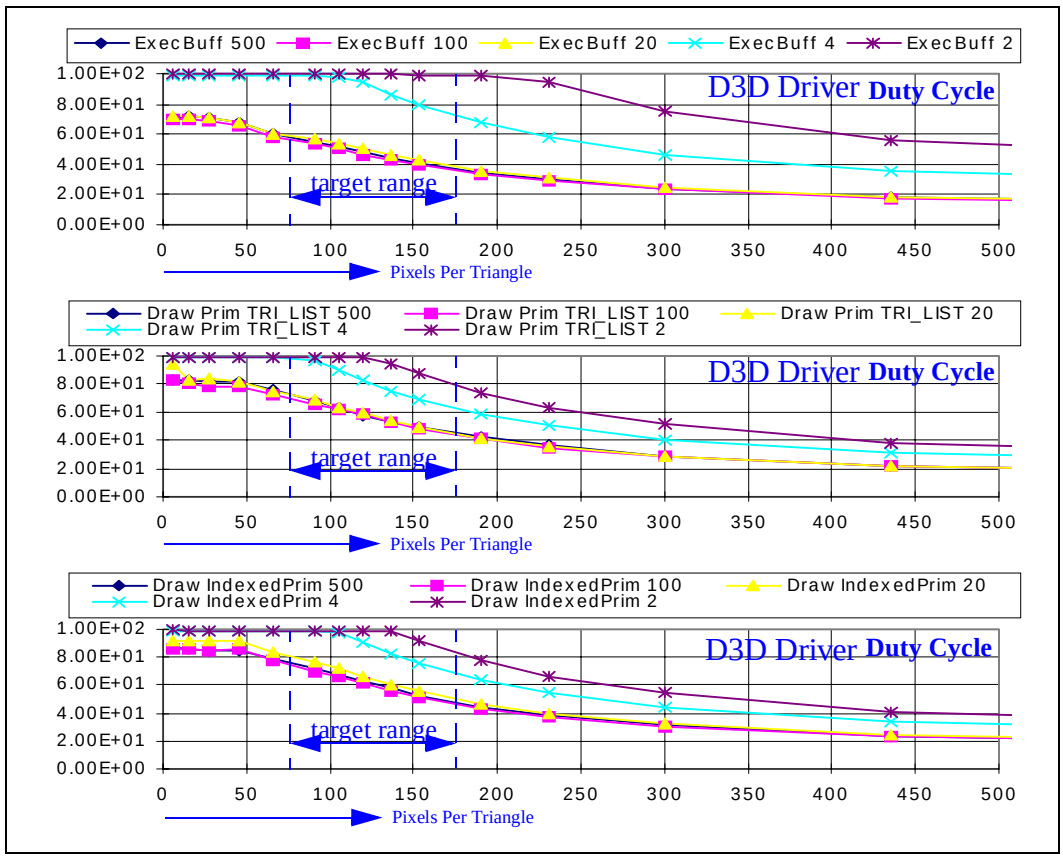
4.1.4.3 Triangle Packet Size

Software designers should try to bunch triangle packets sent to the Intel740™ graphics accelerator driver. Because of the overhead associated with starting the flow of command packets, sending a small number of triangles in a packet decreases performance. By sending out large triangle packets, the overhead is amortized over the rasterization time of all triangles. As a result, higher triangle and fill rates are achieved. Grouping rastered triangles is also critical to maintaining a high level of CPU/Intel740™ graphics accelerator concurrency. For more information on concurrency, see Section 4.1.2, “Using CPU/Intel740™ Graphics Accelerator Concurrency” on page 4-2.

This section addresses both performance vs. execute buffer/draw primitive buffer size, and performance vs. total packet size. The total packet size is the total number of triangles sent between breaks caused by game controls, lighting, or other CPU tasks. It consists of all the execute buffer/ draw primitive buffers sent down one right after the other.

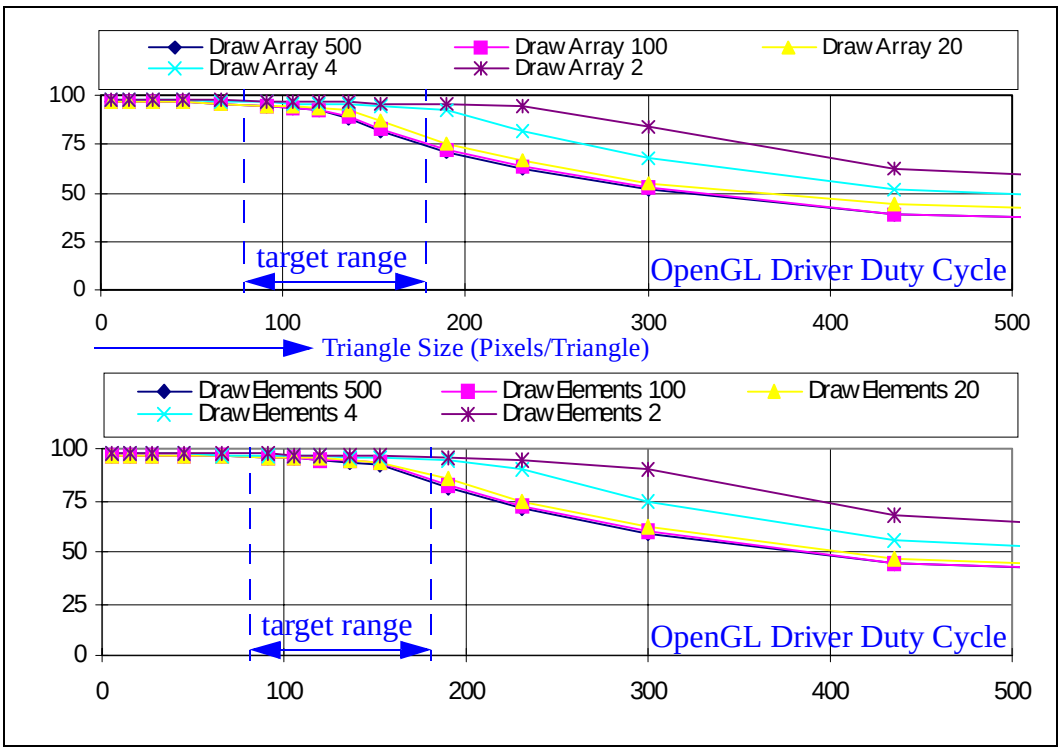
The following graphs illustrate the performance vs. execute buffer size, draw indexed primitive triangle list size, and draw primitive list size. All of these sweeps are full feature sweeps (Fog, 20% Alpha, 30% Specular, MipMap Textures) and have a constant 10,000 triangle total packet size. The Intel740™ graphics accelerator fill rate is not affected; the following graphs show duty cycle (CPU overhead).

Figure 4-11. D3D Performance vs. Buffer Size (Duty Cycle)



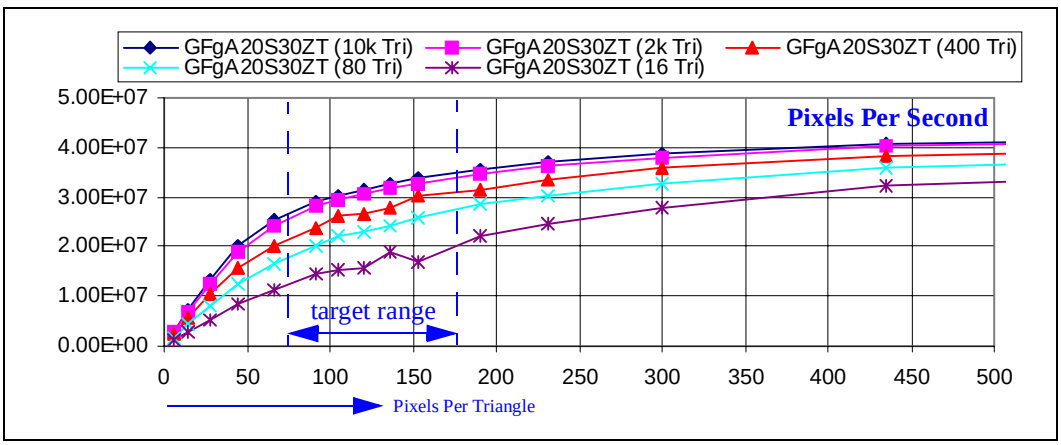
Optimal D3D execute-buffer size on a Pentium® II processor system with an Intel740™ graphics accelerator has been determined to be 512 triangles. Keeping a buffer size above about 50 triangles may be considerably easier to implement and will only cost a few percent performance degradation.

Figure 4-12. OpenGL Performance vs. Buffer Size (Duty Cycle)



The second and equally important concern is performance vs. total packet size. Applications need to have a minimum of about 2,000 triangles per packet (which if organized efficiently is equal to triangles per scene) to achieve near maximum system performance. The following graph illustrates how sending small numbers of triangles in a packet can drastically reduce performance. An example of how this can happen is an application with a render loop which sends many small triangle packets divided up by game controls. Note that the following curves have 100% Z writes in order to keep the %Z constant with changing triangle packet size.

Figure 4-13. Performance vs. Total Packet Size

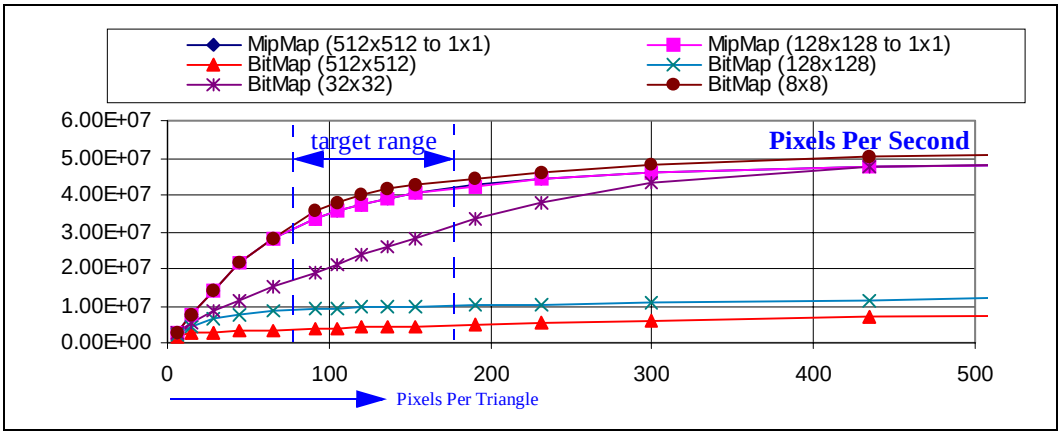


4.1.4.4 Texture Sizes

The ratio of texture-mapped area to triangle area can have a very significant performance impact. Mapping large non-mipmapped textures onto small triangles forces the Intel740™ graphics accelerator to scan through much of the texture for just a few texels. When a textured triangle can be viewed up close as well as far away, mipmapping is an excellent choice. Using mipmapped textures, in addition to looking better, alleviates this problem by selecting a texture map size which is close to the textured triangle size.

The following graph demonstrates how performance can be degraded by texture to triangle size mismatches.

Figure 4-14. Performance vs. Texture Size

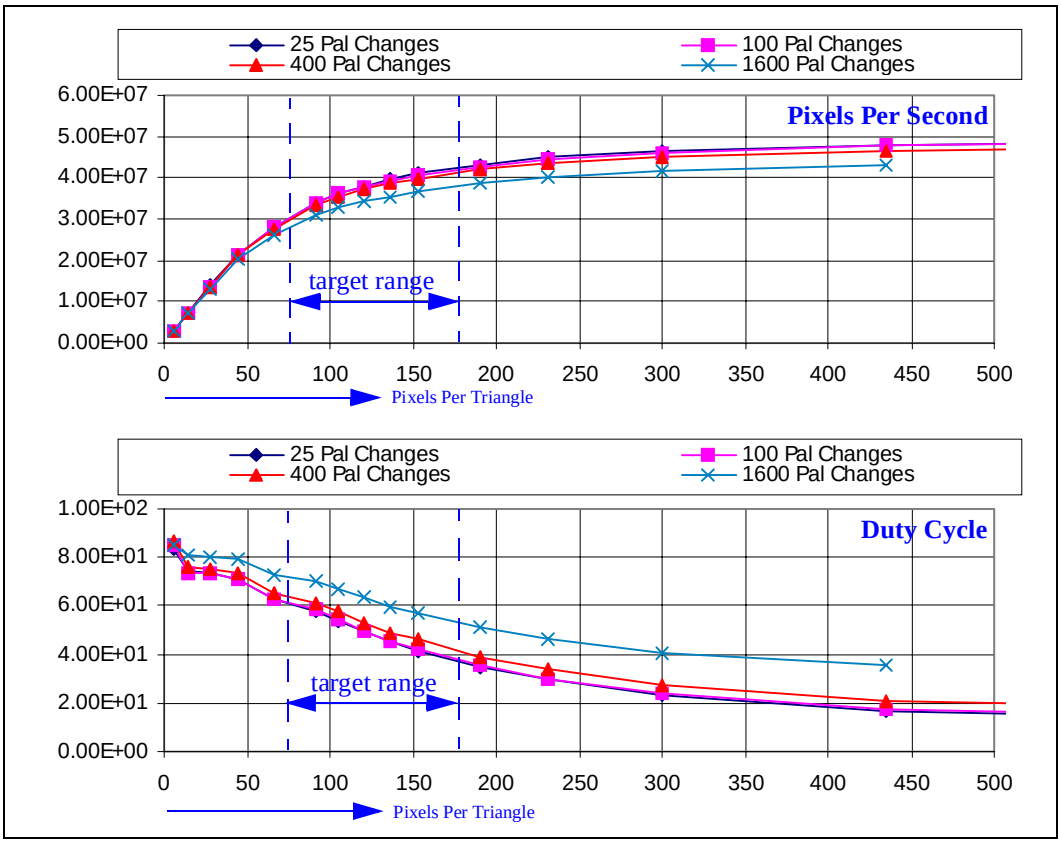


A 32x32 bitmap maps directly onto a 512 pixel triangle. Notice that this size bitmap considerably degrades performance of triangles smaller than about 300 pixels (about half of the direct mapped triangle size). In general, the bitmap area being mapped onto a triangle should be no larger that twice the triangle area in order to maintain high performance. The mipmapped textures (512x512 and 128x128) achieve high performance by allowing the Intel740™ graphics accelerator to select the texture size.

4.1.4.5 Palette Changes

The Intel740™ graphics accelerator is optimized for 16-bit textures. It is recommended that applications use 16-bit textures over 8-bit palettized textures. Palettized textures are supported with a relatively low overhead. The following graph reports the performance of a full-featured scene (Fog, 20% Alpha, 30% Specular, 2.5 DC, 50% Z) with a varied number of palette changes per scene.

Figure 4-15. Performance vs. Palette Changes



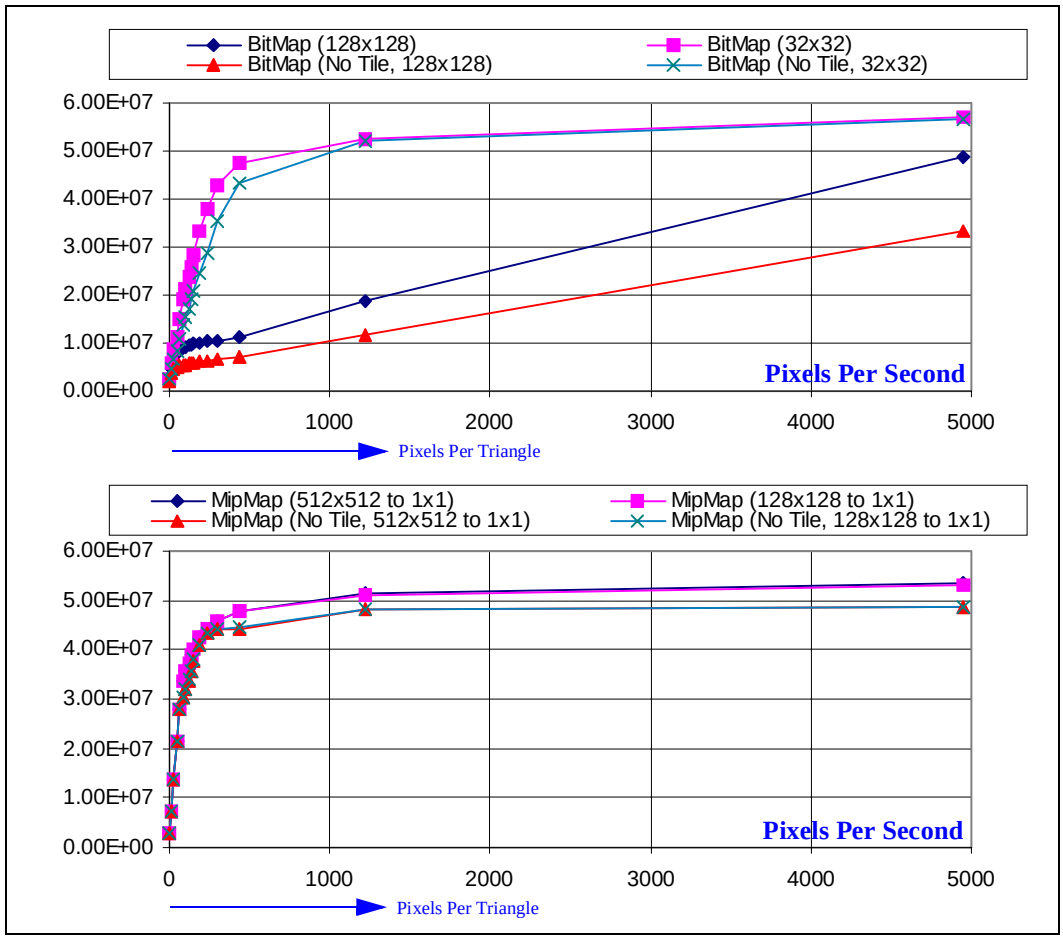
Application developers can use these graphs as an indicator of when to sort palettized textured triangles by texture handle. If an application is CPU limited, sorting by texture handle will degrade performance in most cases.

4.1.4.6 Untiled Textures for Procedural Texture Animation

Directly modifying texture surfaces in AGP memory can be used as a powerful method for creating many types of stunning effects. This section describes the performance implication of using untiled textures. For more information on how to create effects with procedural animation and on Intel740™ graphics accelerator tiling, see Section 4.2.1.4, “Animated Texture Effects” on page 4-22. Note that untiled surfaces can only be created with D3D. OpenGL does not have a mechanism for requesting this type of surface.

Triangles which use untiled textures will be processed with some performance degradation. Figure 4-16 illustrates the performance difference between triangles using tiled and untiled textures.

Figure 4-16. Performance with Untiled Textures



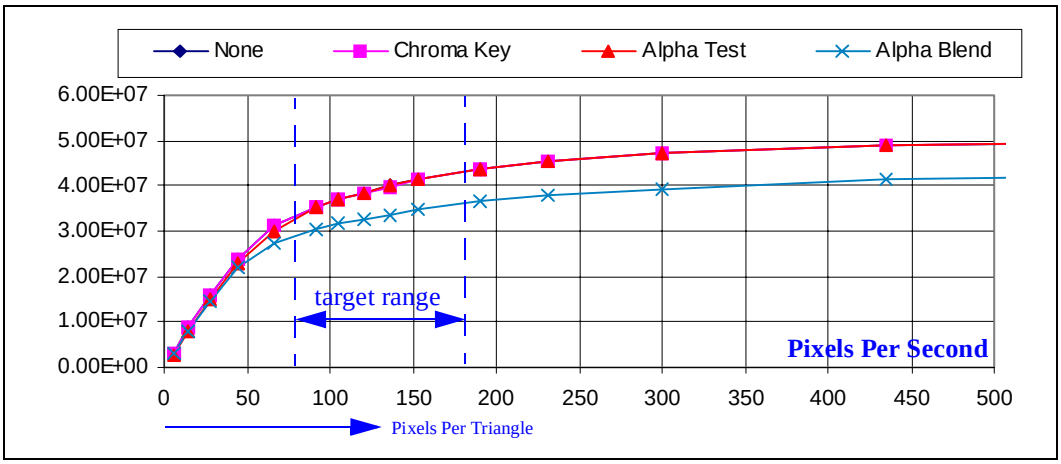
Untiled textures can degrade performance when large texture maps are used or when large triangle to texture map size mismatches are present. Note that in the case of mipmaps, only a small performance degradation is seen for both 512x512 and 128x128. This is because the triangle to texture size mismatch is minimized, so only the degradation with large texture maps is seen. For more information on performance of triangle to texture size mismatch, see Section 4.1.4.4, “Texture Sizes” on page 4-15.

4.1.4.7 High Performance Transparency

Methods of implementing transparency include: chroma keying, alpha testing, and alpha blending. If performance is the primary concern, chroma keying or alpha testing should be used over alpha blending. The Intel740™ graphics accelerator implements both without any performance degradation. When translucency is desired, alpha blending is supported with only a minor performance decrease.

The following graph illustrates the performance of chroma keying, alpha testing, and alpha blending. The sweeps use a feature set of Gouraud, Mipmapped Textures, 2.5 DC, and 50% Z.

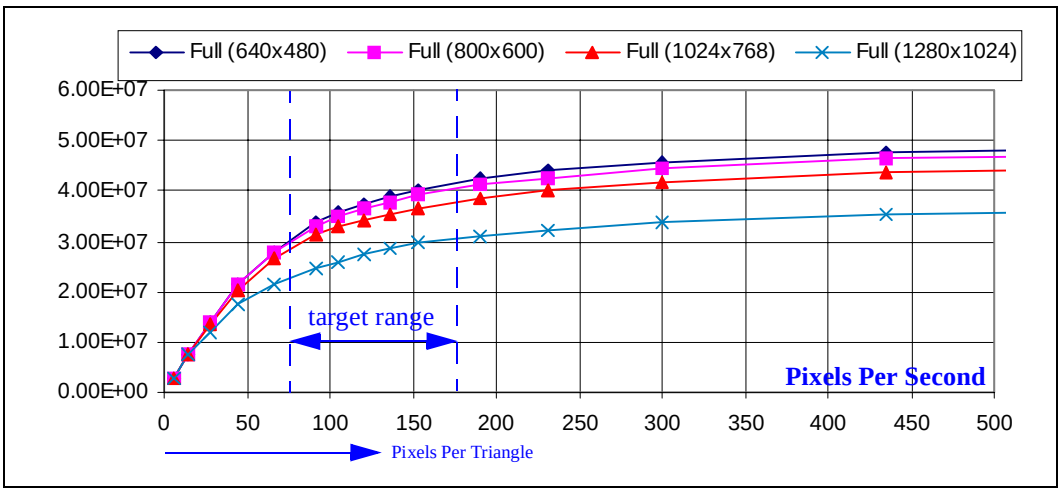
Figure 4-17. Performance vs. Transparency



4.1.4.8 Screen Resolutions

The Intel740™ graphics accelerator 3D performance is optimized for 640x480 and it is recommended that applications target 3D graphic intensive applications for this resolution. The following graph illustrates performance scaling for greater resolutions. The tests are full-feature sweeps (Fog, 20% Alpha, 30% Specular, Mipmapped Textures, 2.5 DC, 50% Z). This test is run with 8 Mbytes of video memory to enable 1280x1024 to fit both the render and Z-Buffer in local video.

Figure 4-18. Performance vs. Screen Resolution



Note that 1280x1024 mode is actually faster than 1024x768 mode because it is interlaced, which does not require as much local memory bandwidth.

4.1.5 Budgeting CPU Clock Cycles

For an application to achieve a sustainable high frame rate, the CPU must calculate lighting, geometry, and game controls, and send the triangle information to the Intel740™ graphics accelerator — all within each frame period. Budgeting CPU clock cycles to fit within the Intel740™ graphics accelerator duty cycle is imperative to this task.

For the Intel740™ graphics accelerator, it is suggested that developers of 3D applications target 10,000 triangles per frame at 30 frames per second. The numbers listed in Figure 4-3 show a conservative analysis of the needed CPU clock cycles and assumptions. The user can anticipate good overall performance when implementing full features of the Intel740™ graphics accelerator and using these targets.

Assumptions:

- Intel740™ graphics accelerator state and operand(s) change overhead not considered
- No Page-Miss on Execute Buffer Reads
- No FP to MMX™ instruction alignment cycles considered
- Theoretical full bandwidth of memory bus available
- Definition of 24 DWords/triangle (96 bytes)

Table 4-3. CPU Cycle Targets

Function	Description	Notes
Frames per Sec	30	
CPU Speed	233 MHz	
CPU Clocks/Triangle	200	
Triangles/Sec	300,000	Triangles/Frame * Frames/Second
Triangles/Frame	10,000	
CPU Clocks/Frame	2,000,000	Triangles/Second * CPU Clocks/Triangle

4.1.6 Video Performance

Figure 4-19. Available Memory Bandwidth on a Pentium® II Processor System

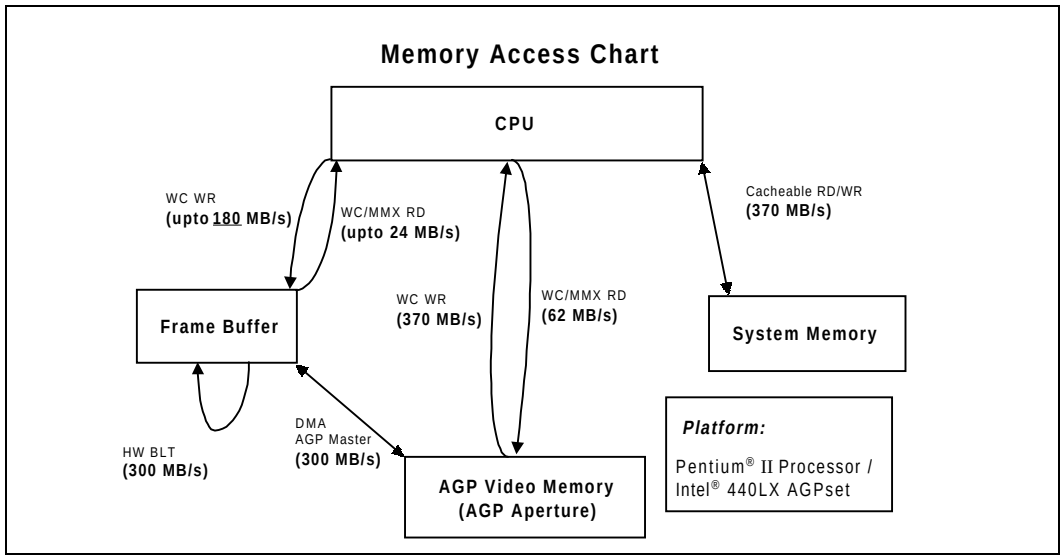


Table 4-4 shows the video/data rates for some typical applications. The highest data rate for video capture is in application of video conferencing on a 200 kbps ISDN line. The highest video display data rate is 20 Mbyte/s in DVD/MPEG-2 playback applications.

Table 4-4. Typical Video/Data Capture Applications

Application	Format	Frame Rate (fps)	Resolution hor*vert*pixdep	Frame Size (bytes)	Bandwidth (Mbytes/s)
Intercast (VBI)	Raw Data	30	800 x 22 x 2	35,200	1.0
POTS Video Conf	Sub-QCIF	15	128 x 96 x 2	24,576	0.37
POTS VC	QCIF	12	176 x 144 x 2	50,688	0.6
ISDN VC (128kbps)	CIF	12	352 x 288 x 2	202,752	2.4
ISDN VC (200kbps)	CIF	15	352 x 288 x 2	202,752	3.0
DVD/MPEG-2	DCIF	30	720 x 480 x 2	691,200	20

Table 4-5 shows the CPU usage for those applications, which can be calculated based on the memory bandwidth. Note that most applications will benefit from the higher read bandwidth of AGP aperture, if the video or VBI data can be routed through the AGP aperture. In this case, the CPU usage for data capturing will be under 5%, making the capture I/O a less degradation factor for the applications. Similarly, the high CPU write bandwidth of AGP aperture can also be useful for DVD/MPEG-2 playback applications.

Table 4-5. CPU Usage for Some Typical Applications

Video/Data Stream			CPU Usage (%)			
Format	Frame Rate	BW (Mbytes/s)	FB Read (BW= 24Mbytes/s)	AGP Read (BW= 62Mbytes/s)	FB Write (BW= 180Mbytes/s)	AGP Write (BW= 360Mbytes/s)
VBI	30fps	1.0	4.2%	1.6%		
SQCIF	15	.37	1.5%	0.6%		
QCIF	12	0.6	2.5%	1.0%		
CIF	12	2.4	10%	3.9%		
CIF	15	3.0	13%	4.8%		
DCIF	30	20			11%	5.6%

4.2 Other Programming Tips

4.2.1 Texture and Surface Effects

Several aspects of texture usage are discussed in this section including:

4.2.1.1 Texture Formats

Because AGP allows high bandwidth for texture execution, and virtually unlimited storage potential (based on the amount of system RAM available) the application developer is no longer limited to small 8-bit palettized textures. There is a whole new world of texture formats which can be experimented with to increase the look and feel of the application. These texture formats are discussed below:

16-bit RGB(A)	Using 16-bit RGB(A) textures is highly recommended because it frees the application from dependence upon the single hardware palette and it allows for a wider span of colors in each texture. Frequent changes of the hardware palette can put a slight strain on the overall performance.
8-bit YUV & 16-bit AYUV	Using YUV textures may provide the user with a new look. When using 8-bit YUV 4:2:2 texture format, storage space is minimized. Also the textures can be input as 16-bit YUV(A) with more colors and more intensities of color as well as alpha. An advantage of YUV is that 8 bits can be represented without the overhead of a palette. YUV is a format that favors the human eye's sensitivity to color because it compresses the chrominance and luminance of a color rather than a degradation between colors. Usually the outcome is a picture which has kept its detail but which is slightly different in color values than the original. Like RGB(A), YUV(A) allows for a much larger range of colors than does a palette.
1, 2, 4, & 8 bit palettized	Sometimes it is good to use a palette for a texture because only a small amount of colors are employed. For instance, if the sky is mainly shades of blue mixed with white, a 4-bit palette could work very well. Because the palette is kept in the hardware, it is not as easy to animate palettes as it is in software. Every time the palette is changed, there is a change in state which causes a performance decrease. This performance decrease is estimated at about 1% if there are 30 palette changes in a frame with 10,000 triangles with full features on.
Live Video Capture	Live Video Capture can be used for a surface texture when combined with 2D Chroma Keying. It might make an astonishing effect if a game incorporated live input of the game player as they are playing the game!

4.2.1.2 Texture Sizes

The Intel740™ graphics accelerator supports texture sizes ranging from 1024 x 1024 to 1 x 1 and any power of two-sided rectangle in between. It is recommended that a few large surface areas take advantage of the large map sizes to show-off this ability where it counts, such as when a background landscape is shown, or to get high resolution detail of a painting. It is best to balance the usage of large and small texture maps to object surfaces that can best utilize them without taking up memory resources when it is not necessary to have that large of a texture.

4.2.1.3 Texture Storage

The Intel740™ graphics accelerator is optimized for texture storage in AGP memory because it allows simultaneous throughput of up to 533 Mbyte/s of textures with the 800 Mbyte/s of local video memory which may contain the display, render and Z-buffer. This equates to 1.3 Gbyte/s total throughput. This is a great advantage over non-AGP graphics accelerator hardware which must keep all the textures in local video memory equating to significantly fewer textures and less local video memory bus bandwidth because it has to share with display, frame and Z-buffers. It is not possible to store textures in local video memory on the Intel740™ graphics accelerator. With DirectX it is as easy to allocate a texture in AGP memory (also known as non-local memory) as it is in local video memory. For OpenGL programmers, the Intel740 drivers automatically place textures into AGP memory when a texture is loaded in the application.

4.2.1.4 Animated Texture Effects

There are many strategies which can be used in animating textures. Each is described below:

UV Coordinates	One way to animate a texture is to change the texture U, V coordinates as they map on to the vertices for each frame. This method of animation is extremely fast since it does not cause any change of state for the hardware and therefore does not cause any performance degradation.
Texture Frames	Several frames can be loaded into AGP for one object and then the object's texture pointer can be changed to cycle through the different textures and give the effect of the textures changing. The drawback is that extra storage space is needed although with more space available for texture storage due to AGP, storing more textures is not a problem.
Specular Lighting	By changing the specular highlighting values along each vertex over time, a change in lighting patterns over an object to simulate water or flickering lights can be produced. The Intel740™ graphics accelerator also allows the Specular Color value to be any R.G.B. color, which means that the colors could be animated to get different effects.
Fogging	As with Specular Highlight animation, Fogging values can be varied over time to produce new and unusual effects such as a whale jumping out of the water and the fog comes off of its body as it hits air and could be replaced with more shininess (specular highlights).
Alpha Blending	By changing the blending factors over time for each frame, the texture can appear more opaque or more translucent over time. This could allow for an effect such as a figure starting out as a ghost object and becoming more visible over time.
Procedural Textures	A procedural texture is one where texel values are changed between renders by a mathematical formula to produce such effects as ripples in water. When creating the texture surface in DirectX, the user needs to specify the DDSCAPS_3DDEVICE so that the surface will not be tiled

if using AGP non local video memory. The texture can be written on by locking the texture surface and getting a pointer to it. Then the user can traverse the texture memory space and apply their changes. This is a great way to represent fire or water in a texture and utilizes the extra CPU cycles while scene rendering is being done by the hardware. Textures should be stored in AGP memory to take advantage of the Direct Memory Execution (DME) abilities of the Intel740™ graphics accelerator. For OpenGL programmers, there is no way to specify "3DDEVICE" which means there will be a performance penalty when using procedural textures because the Intel740 drivers will tile them each time they are loaded. Also with OpenGL, there is no way to access the texture memory through the API so the image data will need to be updated from its source and then reloaded through the API after each change of the texture for each frame that is dependent on that texture surface.

4.2.1.5 Multi-pass Texture Effects

There are a few more texture effects worth mentioning that can be obtained with multi-pass algorithms. Multi-pass means that the scene is rendered twice for each frame, hence causes a 2x slower performance. The different effects are listed below:

Z-Buffer Shadowing — First the camera must view the scene from the point of view of a light source. The scene is rendered using Z-buffering. On the second pass, the scene is rendered from the real point of view of the camera, and then the old Z-buffer values are read with a color altering algorithm which is applied to the pixel values being rendered at the same x, y location, thus creating a shadow.

Dual Texture Rendering — The first scene is rendered with textures in their correct places, then the second scene is rendered with a common pattern (possibly using one of the animated texture techniques) such as a translucent lighting effect. In this way underwater rocks, plants, and animals could all appear to be affected by the same light patterns.

4.2.2 Software Strategies

This section describes how to optimize applications which take advantage of the many features of the Intel740™ graphics accelerator. Topics of discussion include:

- Using Z-Buffering
- Using Triple-Buffering
- Using Antialiasing
- Minimizing State Transitions
- Using Dynamic AGP Buffer Placement
- Using Texture Palettes
- Using Mipmapping
- Optimal Artist Geometry Design
- Optimal Artist Texture Design for Trilinear Filtering
- Using Color/Chroma keying on Top of Alpha Blended Textures
- Avoiding Stippling Errors
- Avoiding Flipping Errors
- Texture Sorting is Not Required

4.2.2.1 Using Z-Buffering

The Intel740™ graphics accelerator performs all of an application's 3D depth compare in the hardware. This means that the hardware will correctly write all of the triangles in the scene as they overlap, without the need for breaking them up into smaller triangles or expensive sorting algorithms. What the programmer must remember is that if the polygons (triangles) are sorted from back to front in the application and then sent to the hardware with the Z-buffering enabled, this will give worse case results because the hardware Z-buffer algorithm checks each pixel in an x, y position against the last, and if it is in front of the last according to its z value, it will write over it. It is best not to sort at all if the Z-buffer is enabled. However, enabling anti-aliasing or alpha blending requires that the triangles be sorted from back to front. In this case Z-buffering may cause a performance hit which becomes a trade-off for rendering any intersecting triangles properly.

The Intel740™ graphics accelerator supports a 16-bit Z-buffer. Sometimes an application's scene depth complexity will cause rounding of the z bits resulting in unwanted tears along some polygons. To alleviate this problem the user could disable Z-buffering for background items and render them first. Another solution is to make the scene's z coordinates fit within a 16-bit range.

4.2.2.2 Using Triple-Buffering

It is highly recommended to implement triple-buffering for fullscreen applications. There are a couple of reasons to implement triple-buffering. First, many fullscreen applications experience a stall while the driver waits for the vsynch signal so it can flip from the back to the primary buffer for each frame of the application. In such instances, triple-buffering will minize the "wait for flip" stall because the application can draw to a second back buffer which will not have the vsynch dependency associated with it. Also, because the Intel740™ graphics accelerator textures out of AGP memory, the local video memory does not have to be shared and so more buffers can be created in local video memory without any loss of texturing capability.

4.2.2.3 Using Antialiasing

It is very easy to implement anti-aliasing. Simply enable it. Sort the polygons/triangles back to front, and render the scene. The user should be cautioned to use anti-aliasing sparingly as it causes a performance slow down. The user should also note that anti-aliasing requires that Z-buffering be enabled and that a Z function is defined. One strategy for rendering a scene with anti-aliasing and Z-buffering acceleration would be to render the background separately without anti-aliasing or Z-buffering enabled, then sort back to front the forefront items, enable both anti-aliasing and Z-buffering and then render the rest of the scene.

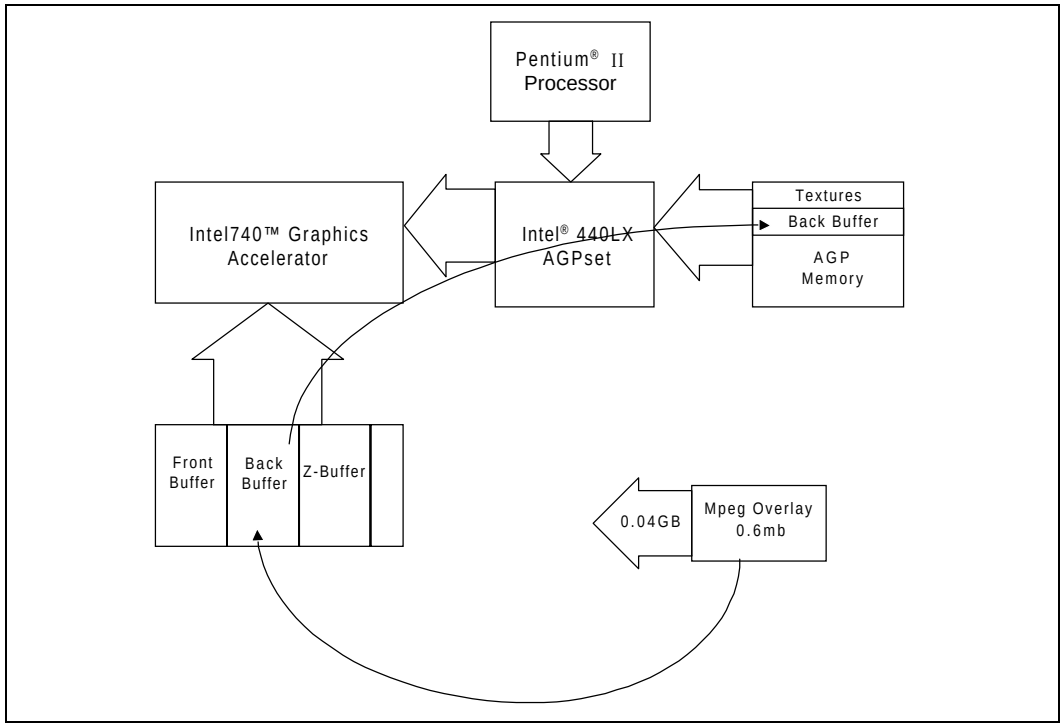
4.2.2.4 Minimizing State Transitions

It is encouraged that as much of the features of the Intel740™ graphics accelerator be utilized during the execution of a 3D program as is needed to achieve the maximum visual effect. There is little overhead for enabling all of the Intel740™ graphics accelerator 3D features with the exception of alpha blending and anti-aliasing which should only be enabled as needed. Each time a feature is enabled or disabled, a state change must take place within the hardware. State changes cause a slight decrease in overall bandwidth and so causes a slight performance hit. Best performance will be ensured if triangles to be rendered are ordered according to their state or the set of features they have enabled. For the most part, state changes do not affect the Intel740™ graphics accelerator. The only state changes which cause a pipeline flush are palette and stippling changes.

4.2.2.5 Dynamic AGP Buffer Placement

The Intel740™ graphics accelerator supports dynamic AGP Buffer Placement. Alternate buffers can be relocated from local video memory into AGP memory when necessary to allow full functionality. When there is 2 Mbytes of local video memory, at 640 x 480 x 16 the front buffer, back buffer and Z-buffer can all be placed in local video memory. When the resolution is changed to 800 x 600 x 16 or higher, then the back buffer can be relocated to AGP memory. The Intel740™ graphics accelerator supports rendering to the back buffer in AGP memory. Putting the back buffer into AGP memory can free up local video memory for MPEG Overlay as well.

Figure 4-20. Dynamic AGP Buffer Placement



4.2.2.6 Using Texture Palettes

It is best not to use palettized textures, because the Intel740™ graphics accelerator supports many formats of ARGB, YUV and AYUV, which allows more colors without the overhead of palette loads and changes. To use palettized textures, minimize changes in palettes. The hardware only supports one palette and to change it requires a state change and a pipeline flush, which slows overall performance. There are ways to combine many texture palettes into one, with the use of a tool such as Debabelizer* which can find a common palette among many textures. It is best to use texture formats that require no palette at all.

4.2.2.7 Using Mipmapping

An application not only increases visual quality but can also increase performance of the application by using mipmapped textures. When an object becomes very small or distant and it has a large texture map associated with it, the ratio of texel look-up to texels used in rendering can be 8 to 1 because the Intel740™ graphics accelerator drivers are acquiring 16 bytes from a section of the texture map but only 2 bytes are actually being rendered. Mipmapping will give a 1 to 1 ratio of texture texels read from an image to those texels rendered in the scene. Mipmapping usually improves overall application performance by at least 10%.

Mipmapping provides better looking graphical representation of a scene by allowing the user to create various texture maps, which the hardware will choose to map onto the object based on how far the object is from the viewer. So if a scene has a patterned texture which is mapped onto an object, the user would want to create variations of that pattern which would get smaller and smaller to correspond with each mipmapping level. The user then sets a pointer to each level of mipmap so that the hardware will choose the correct texture based on the distance from the viewer. The Intel740™ graphics accelerator supports tri-linear mipmapping for added visual quality.

4.2.2.8 Optimal Artist Geometry Design

Improper geometry creation causes application anomalies but can be completely avoided when a few good geometry creation techniques are implemented. One geometry problem is caused when two objects are intersecting and when their individual vertices overlap or when two object's vertices are very close together without connection. When the Z values for these overlapping vertices are less than a 2^16 step from each other, both of the vertices will have the same Z value due to rounding causing the hardware to choose either pixel to be drawn. The result is referred to as "pixel popping". One way to avoid pixel popping is to make sure that overlapping objects share vertices and edges at the point of intersection. The shared vertices need to be precisely the same value in order for the solution to work. The second geometry problem is caused when a vertice forms a "T" side where three triangles come together and do not share a common vertice. Where the common vertice is not shared, "texture tearing" occurs due to precision pixel interpolation done in the hardware. The solution then, is to avoid creating geometry "T"s.

4.2.2.9 Optimal Artist Texture Design for Trilinear Filtering

Trilinear filtering with the Intel740™ graphics accelerator looks best when the texture mipmap levels are created using a "nearest" value algorithm rather than an averaged value algorithm. When the artist creates the texture levels, they can control the type of filtering used to create the smaller sizes with their texture creation tools. Trilinear filtering with the Intel740 graphics accelerator adds another level of filtering from the bilinear method so, in some instances, textures can appear to become blurred. Sometimes the artist may like the extra blending as in the case of chromakeyed trees and shrubs where the blended edges add a more natural appearance. In most instances, especially when text is placed in a texture such as on a roadway sign, the images will need to be as sharp as possible so they can be understood from far away.

4.2.2.10 Using Color/chroma Keying on Top of Alpha Blended Textures

When using both alpha blending and chroma/colorkeying together in a rendered frame there are some renderstates which must be enabled to ensure that all textures are rendered properly. Use the following DirectX render states:

```
SetRenderState(D3DRENDERSTATE_ALPHATESTENABLE, TRUE);
SetRenderState(D3DRENDERSTATE_ALPHAFUNC, D3DCMP_NOTEQUAL);
SetRenderState(D3DRENDERSTATE_ALPHAREF, 0);
```

At the same time, chroma/color keying should also be enabled using the DirectX function, SetColorKey() and setting the dwColorSpaceLowValue and dwColorSpaceHighValue properly. Remember that for color keying, both values should be the same color palette index value, and for chroma keying, the both values should be the same value for high and low as to how the RGB has been defined.

4.2.2.11 Avoiding Stippling Errors

Some developers have set D3DRENDERSTATE_STIPPLEENABLE to TRUE which sets the default value of 0 to be set for all stippled patterns from D3DRENDERSTATE_STIPPLEPATTERN00 to D3DRENDERSTATE_STIPPLEPATTERN31. The result of enabling stippling without setting any values will be a black screen since all of the values are by default set to zero. If developers are not going to be using stippling, they should not use this render state at all. If they are going to use stippling, they should be sure to set the stippling values for all the D3DRENDERSTATE_STIPPLEPATTERN*XX*. When stippling is not to be in use, developers should make sure to set D3DRENDERSTATE_STIPPLEENABLE to FALSE.

4.2.2.12 Avoiding Flipping Errors

When using the DirectX API, it is important to always use the BeginScene and EndScene calls at the beginning and end of each frame to be written. These calls ensure that flipping errors such as blanking screens do not occur.

4.2.2.13 Texture Sorting Is Not Required

With the Intel740™ graphics accelerator, the user does not have to sort textures because even though changing the texture pointer is a state change, it does not cause a pipeline flush and will not noticeably slow down the rendering. The application would be much slower at sorting textures than the Intel740™ graphics accelerator would be at swapping handles. If texture sorting for static geometry can be done once to affect many frames, it might be useful to do so. If palettized textures are used, a performance hit may result because each pixel written could change palettes many times when relying on hardware Z-buffering for sorting. Because hardware Z-buffering will always be faster than software sorting algorithms, it is recommended that the user move toward RGBA or YUV textures, which will not have a performance impact.

4.3 OpenGL Programming Implementation

The Intel740™ graphics accelerator supports all OpenGL commands and parameters as specified in *The OpenGL Graphics System: A Specification*. This requires the OpenGL implementation to be divided between the CPU and the graphics subsystem, in varying degrees according to the operations involved and the functionality and performance of those system components. This characteristic of OpenGL implementations is desirable because the application is not required to understand the division of labor (and its resultant performance).

In many instances, the performance of a software implementation cannot be tolerated because minimum frame rates cannot be attained. This document specifies which functions/features of OpenGL V1.1 will be hardware-accelerated (vs. performed in software or require software rasterization) by the Intel740™ graphics accelerator OpenGL implementation. By using accelerated features and avoiding software rasterization, a developer can gain some assurance that the application will run at a high level of performance. An application still needs to be tuned to ensure the highest level of performance. That the Intel740™ graphics accelerator OpenGL implementation is “complete” and contains all the required functionality.

4.3.1 OpenGL Feature Classification

For the Intel740™ graphics accelerator OpenGL implementation, OpenGL “features” fall into three categories:

1. Features supported directly by graphics hardware (such as setup and most per-fragment operations). These features are implemented through hardware acceleration and are classified by “HW Accelerated.”
2. Features not supported by graphics hardware which would require software rasterization (such as stencil operations) are implemented through the generic OpenGL software emulation and are classified by “SW Emulation.” These features should be used sparingly.
3. “CPU-supported” features (geometry, lighting, display lists, etc.) which, although not particularly accelerated by graphics hardware, are likely to provide a level of performance equal to (or greater) than similar functions performed in the application. These features are implemented through a combination of hardware acceleration and software emulation and are classified by “HW/SW Hybrid” and their usage is not necessarily detrimental to performance.

Note: The programmer must consider all the pertinent state variables in order to understand what will be hardware accelerated — a single mode might preclude acceleration of all primitive rasterization.

4.3.2 Feature Overview

The following table lists (at a high level) the rating of the OpenGL features.

Table 4-6. Rating OpenGL Features (Sheet 1 of 2)

Function	Classification†	Comments
Pixel Formats		
RGBA	HW Accelerated	
Color Index	SW Emulation	
Vertex Specification		
Begin/End	HW/SW Hybrid	
Vertex Array	HW/SW Hybrid	
Evaluator	HW/SW Hybrid	
Model-view Transform	HW/SW Hybrid	
Lighting	HW/SW Hybrid	
Texture Generation	HW/SW Hybrid	
Texture Transform	HW/SW Hybrid	
User Clip Planes	HW/SW Hybrid	
Projection Transform	HW/SW Hybrid	
View Volume Clipping	HW/SW Hybrid	
Perspective Divide	HW/SW Hybrid	
Viewport Transform	HW/SW Hybrid	
Current Raster Position	HW/SW Hybrid	
Pixel Operations	SW Emulation	
Point Rasterization		
Width	HW/SW Hybrid	
Anti-aliasing	SW Emulation	
Line Rasterization		
Width	HW Accelerated/SW Emulation	HW Accelerated: Width = 1.0
Smoothing	HW Accelerated	
Stippling	HW Accelerated/SW Emulation	HW Accelerated: for trivial patterns
Polygon Rasterization		
Culling	HW Accelerated	
Stippling	HW Accelerated	
Smoothing	HW Accelerated	
Fill Mode	HW Accelerated	
Point Mode	HW/SW Hybrid	
Line Mode	HW Accelerated	
Depth Offset	HW/SW Hybrid	
Pixel Rectangles / Bitmaps	HW Accelerated	HW Accelerated: simple copy operations

† See also Section 4.3.1.

Table 4-6. Rating OpenGL Features (Sheet 2 of 2)

Function	Classification†	Comments
Texturing		
TexImage*	HW/SW Hybrid	
CopyTex	HW/SW Hybrid	
TexSubImage	HW/SW Hybrid	
CopyTexSubImage	HW/SW Hybrid	
Wrap	HW Accelerated	
Bilinear Filtering	HW Accelerated	
Trilinear Filtering	SW Emulation	
Border	SW Emulation	
Texture Objects	HW/SW Hybrid	
Replace, Modulate, Decal Modes	HW Accelerated	
Blend Mode	SW Emulation	
Fog		
Per-Vertex	HW Accelerated	
Per-Pixel	SW Emulation	
Per-Fragment Operations		
Pixel Ownership	SW Emulation	when drawing to occluded front buffer
Scissor	SW Emulation	
Alpha Test	HW Accelerated	
Stencil	SW Emulation	
Depth Buffer Test	HW Accelerated	
Blending	HW Accelerated	Note: No destination alpha buffer with depth buffer
Dithering	HW Accelerated	
Logical Op	SW Emulation	except for trivial operations
Whole Frame Buffer Operations		
FRONT_AND_BACK	HW/SW Hybrid	driver must draw twice
Stereo Buffers	n/a	Not supported
Auxiliary Buffers	n/a	Not supported
Buffer Masks	HW Accelerated/SW Emulation	SW Emulation: different R,G,B,A masks
Clear	HW Accelerated	
Accumulation Buffer	HW/SW Hybrid	accumulation performed in software
Read Pixels	HW/SW Hybrid	
Copy Pixels	HW Accelerated/SW Emulation	HW Accelerated: simple copies
Selection	HW/SW Hybrid	
Feedback	HW/SW Hybrid	
State Requests	HW/SW Hybrid	

† See also Section 4.3.1.

Note: The remainder of this chapter is structured as an “annotation” of the OpenGL V1.1 specification and specific extensions. Only performance notes will be discussed and included here, so one probably needs to read this document alongside the OpenGL specification.

4.3.3 OpenGL Operation

The following sections describe the classification of OpenGL features.

4.3.3.1 Begin/End Paradigm

There are no primitive (object) types excluded from hardware acceleration. Points, line segments, line segment loops, separated line segments, polygons, triangle strips, triangle fans, separated triangles, quadrilateral strips, and separated triangles are all candidates for hardware acceleration. This includes the specification of polygon edge flags.

4.3.3.2 Vertex Specification

All vertex and associated auxiliary data specifications are included in the performance set, with the following exceptions:

Since color index mode is not supported. Index specification is not of particular interest

4.3.3.3 Vertex Arrays

Vertex array specification is included in the performance set, and is the preferred means to describe objects with a large number of vertices.

4.3.3.4 Rectangles

Rectangles are included in the performance set.

4.3.3.5 Coordinate Transformation

The Intel740™ graphics accelerator does not provide hardware acceleration for transformations, although vertex, normal, and texture coordinate transformations are supported and optimized for the target platform. These operations are therefore rated PG.

Application designers wishing to perform these operations internally are referred to the “OpenGL Correctness Tips” provided in the *OpenGL Programming Guide;* directions are given to allow 2D rasterization specification. Note that the viewport transformation is always enabled and thus must be set to properly generate the proper window coordinates.

4.3.3.6 Clipping

The Intel740™ graphics accelerator OpenGL implementation does not provide hardware acceleration for view-volume or client clip plane clipping. These operations will require a software clipping stage prior to rasterization.

4.3.3.7 Current Raster Position

Not all operations which rely on the current raster position are hardware accelerated.

4.3.3.8 Colors and Coloring

The Intel740™ graphics accelerator does not provide hardware accelerated lighting operations. Although lighting is supported, applications wishing to perform these operations internally should ensure that lighting is disabled in OpenGL.

Both flat shading modes (SMOOTH and FLAT) are supported by the Intel740™ graphics accelerator hardware.

4.3.4 Rasterization

4.3.4.1 Antialiasing

Line and polygon smoothing is supported by the Intel740™ graphics accelerator hardware.

4.3.4.2 Points

Aliased points are rendered by the Intel740™ graphics accelerator hardware using short lines or triangles. Antialiased points will require software rasterization.

4.3.4.3 Line Segments

Only unit-width aliased or anti-aliased lines are supported by the Intel740™ graphics accelerator hardware. Stippled and/or wide lines are not supported by the hardware and will require a software or hybrid rasterization phase.

4.3.4.4 Polygons

Polygon culling is supported by the Intel740™ graphics accelerator hardware, as are stippled and/or anti-aliased polygons.

FILL and LINE polygon modes are supported by the Intel740™ graphics accelerator hardware. Depth offset is not directly supported by the hardware, but does not require software rasterization.

4.3.4.5 Pixel Rectangles

Pixel rectangles are not supported by the Intel740™ graphics accelerator hardware and will require software rasterization.

4.3.4.6 Bitmaps

Bitmaps are not supported by the Intel740™ graphics accelerator hardware and will require software rasterization.

4.3.4.7 Texturing

All texture mapping functions are supported by the Intel740™ graphics accelerator hardware, with the following exceptions:

- Border colors are ignored (textures are clamped to the edges)
- BLEND texture function requires software rasterization

4.3.4.8 Fog

The Intel740™ graphics accelerator hardware supports linear interpolation of the fog factor. Setting the FOG_HINT to NICEST when EXP or EXP2 modes are selected will require software rasterization.

4.3.4.9 Antialiasing Application

Line and polygon smoothing is supported by the Intel740™ graphics accelerator hardware.

4.3.5 Fragments And The Frame Buffer

4.3.5.1 Per-Fragment Operations

The following table defines which pre-fragment operations are included or excluded from the performance set.

Table 4-7. Included and Excluded Pre-Fragment Operations

Operation	Classification†
Pixel Ownership	SW Emulation (when drawing to an occluded front buffer)
Scissor	SW Emulation
Alpha Test	HW Accelerated
Stencil	SW Emulation
Depth Buffer Test	HW Accelerated
Blending	HW Accelerated, though destination alpha buffer is not supported
Dithering	HW Accelerated
Logical Operation	SW Emulation

† See also Section 4.3.1.

4.3.5.2 Whole Framebuffer Operations

Drawing to the FRONT_AND_BACK will require two rasterization passes (internal to the OpenGL implementation). Stereo and auxiliary buffers are not supported.

Use of ColorMask should be limited to enabling or disabling all the color components concurrently. Software rasterization will be required if only some of the color components masked.

Those “whole frame buffer” operations related to stencil or accumulation buffers will require software rasterization.

4.3.5.3 Drawing, Reading, and Copying Pixels

Only “pure” copy pixel operations are hardware accelerated. Pixel reads will be performed in software.

4.3.6 Special Functions

The special functions (listed below) are all performed by the CPU and are therefore rated “HW/SW Hybrid.”

- Display lists
- Flush and Finish
- Evaluators
- Selection
- Feedback

4.3.7 State And State Requests

All of the state request commands are performed in software are therefore rated “HW/SW Hybrid.”

4.3.8 GL Command Summary

The following table provides “performance ratings” on a per-command basis, with notes on parameter settings.

Table 4-8. Command Performance Ratings (Sheet 1 of 5)

Command/Feature	Classification†	Comment/Exception
glAccum	HW/SW Hybrid	
glAlphaFunc	HW Accelerated	
glAreTexturesResident	HW/SW Hybrid	
glArrayElement	HW/SW Hybrid	
glBegin	HW/SW Hybrid	
glBindTexture	HW/SW Hybrid	
glBitmap	SW Emulation	
glBlendFunc	HW Accelerated	
glCallList	HW/SW Hybrid	
glCallLists	HW/SW Hybrid	
glClear	HW Accelerated	
glClearAccum	SW Emulation	
glClearColor	HW Accelerated	
glClearDepth	HW Accelerated	
glClearIndex	SW Emulation	color index not supported
glClearStencil	SW Emulation	
glClipPlane	HW/SW Hybrid	requires software clipping
glColor	HW/SW Hybrid	
glColorMask	HW Accelerated/SW Emulation	HW Accelerated: only when all channels are masked or enabled together

† See also Section 4.3.1.

Table 4-8. Command Performance Ratings (Sheet 2 of 5)

Command/Feature	Classification†	Comment/Exception
glColorMaterial	HW/SW Hybrid	
glColorPointer	HW/SW Hybrid	
glCopyPixels	HW Accelerated/SW Emulation	HW Accelerated: for simple copies
glCopyTex*	HW Accelerated/SW Emulation	HW Accelerated: for simple copies
glCullFace	HW Accelerated	
glDeleteLists	HW/SW Hybrid	
glDeleteTextures	HW/SW Hybrid	
glDepthFunc	HW Accelerated	
glDepthMask	HW Accelerated	
glDepthRange	HW/SW Hybrid	
glDisable	-	see glEnable
glDisableClientState	HW/SW Hybrid	
glDrawArrays	HW/SW Hybrid	
glDrawBuffer	HW Accelerated	HW Accelerated: only NONE, FRONT or BACK
glDrawElements	HW/SW Hybrid	
glDrawPixels	HW Accelerated/SW Emulation	HW Accelerated: for simple copies
glEdgeFlag	HW Accelerated	
glEdgeFlAGPointer	HW/SW Hybrid	
glEnable		
*_ARRAY	HW/SW Hybrid	
ALPHA_TEST	HW Accelerated	
AUTO_NORMAL	HW/SW Hybrid	
BLEND	HW Accelerated/SW Emulation	SW Emulation: destination alpha buffer not supported
CLIP_PLANEi	HW/SW Hybrid	
COLOR_MATERIAL	HW/SW Hybrid	
CULL_FACE	HW Accelerated	
DEPTH_TEST	HW Accelerated	
DITHER	HW Accelerated	
FOG	HW Accelerated/SW Emulation	SW Emulation: when FOG_HINT == NICEST and not LINEAR fog
LIGHTi	HW/SW Hybrid	
LIGHTING	HW/SW Hybrid	
LINE_SMOOTH	HW Accelerated	
LINE_STIPPLE	HW Accelerated/SW Emulation	HW Accelerated: trivial patterns

† See also Section 4.3.1.

Table 4-8. Command Performance Ratings (Sheet 3 of 5)

Command/Feature	Classification†	Comment/Exception
*_LOGIC_OP	HW Accelerated/SW Emulation	HW Accelerated trivial operations
MAP*	HW/SW Hybrid	
NORMALIZE	HW/SW Hybrid	
POINT_SMOOTH	SW Emulation	
POLYGON_OFFSET*	HW/SW Hybrid	
POLYGON_SMOOTH	HW Accelerated	
POLYGON_STIPPLE	HW Accelerated	
SCISSOR_TEST	SW Emulation	
STENCIL_TEST	SW Emulation	
TEXTURE_*D	HW/SW Hybrid	
TEXTURE_GEN*	HW/SW Hybrid	
glEnd	-	
glEndList	HW/SW Hybrid	
glEval*	HW/SW Hybrid	
glFeedbackBuffer	HW/SW Hybrid	
glFinish	HW/SW Hybrid	
glFlush	HW/SW Hybrid	
glFog	HW Accelerated/SW Emulation	SW Emulation: when FOG_HINT == NICEST and not LINEAR
glFrontFace	HW Accelerated	
glFrustrum	HW/SW Hybrid	
glGenLists	HW/SW Hybrid	
glGenTextures	HW/SW Hybrid	
glGet*	HW/SW Hybrid	
glHint	-	depends on hint
glIndex*	SW Emulation	color index not supported
glInitNames	HW/SW Hybrid	
glInterleavedArrays	HW/SW Hybrid	
glIs*	HW/SW Hybrid	
glLight	HW/SW Hybrid	
glLightModel	HW/SW Hybrid	
glLineStipple	HW Accelerated/SW Emulation	HW Accelerated: when solid
glLineWidth	HW Accelerated/SW Emulation	HW Accelerated: when 1.0
glListBase	HW/SW Hybrid	
glLoadIdentity	HW/SW Hybrid	
glLoadMatrix	HW/SW Hybrid	

† See also Section 4.3.1.

Table 4-8. Command Performance Ratings (Sheet 4 of 5)

Command/Feature	Classification†	Comment/Exception
glLoadName	HW/SW Hybrid	
glLogicOp	HW Accelerated/SW Emulation	HW Accelerated: when CLEAR, COPY or SET
glMap*	HW/SW Hybrid	
glMaterial	HW/SW Hybrid	
glMatrixMode	HW/SW Hybrid	
glMultMatrix	HW/SW Hybrid	
glNewList	HW/SW Hybrid	
glNormal	HW/SW Hybrid	
glNormalPointer	HW/SW Hybrid	
glOrtho	HW/SW Hybrid	
glPassThrough	HW/SW Hybrid	
glPixelMap	HW Accelerated/SW Emulation	HW Accelerated: trivial operations
glPixelStore	HW Accelerated/SW Emulation	HW Accelerated: trivial operations
glPixelTransfer	HW Accelerated/SW Emulation	HW Accelerated: trivial operations
glPixelZoom	SW Emulation	
glPointSize	HW Accelerated/ HW/SW Hybrid	HW Accelerated: only for unit width
glPolygonMode	HW Accelerated/ HW/SW Hybrid	HW Accelerated: when FILL or LINE
glPolygonOffset	HW/SW Hybrid	
glPolygonStipple	HW Accelerated	
glPopAttrib	HW/SW Hybrid	
glPopMatrix	HW/SW Hybrid	
glPopName	HW/SW Hybrid	
glPrioritizeTextures	HW/SW Hybrid	
glPushAttrib	HW/SW Hybrid	
glPushMatrix	HW/SW Hybrid	
glPushName	HW/SW Hybrid	
glRasterPos	HW/SW Hybrid	
glReadBuffer	SW Emulation	
glReadPixels	SW Emulation	
glRect	HW Accelerated	
glRenderMode	HW Accelerated/ HW/SW Hybrid	HW Accelerated: RENDER; HW/SW Hybrid: SELECT or FEEDBACK
glRotate	HW/SW Hybrid	
glScale	HW/SW Hybrid	

† See also Section 4.3.1.

Table 4-8. Command Performance Ratings (Sheet 5 of 5)

Command/Feature	Classification†	Comment/Exception
glScissor	SW Emulation	
glSelectBuffer	HW/SW Hybrid	
glShadeModel	HW Accelerated	
glStencil*	SW Emulation	
glTexCoord	HW Accelerated	
glTexEnv	HW Accelerated/SW Emulation	SW Emulation: BLEND
glTexGen	HW/SW Hybrid	
glTexImage1d	HW/SW Hybrid	border colors ignored
glTexImage2d	HW/SW Hybrid	border colors ignored
glTexParameter	HW Accelerated/SW Emulation	SW Emulation: *_MIPMAP_LINEAR and TEXTURE_BORDER_COLOR
glTextureSubImage*	HW/SW Hybrid	
glTranslate	HW/SW Hybrid	
glVertex	HW/SW Hybrid	
glVertexPointer	HW/SW Hybrid	
glViewport	HW/SW Hybrid	

† See also Section 4.3.1.

Appendix A. Creating a VPE Port Sample A

VPE.H File Listing

```
// File Name: vpe.h

#include “ddraw.h”
#include <dvp.h>

#define OVERLAY_MAX_WIDTH 720
#define OVERLAY_MAX_HEIGHT 1024
#define YUY2_4CC mmioFOURCC(‘Y’,’U’,’Y’,’2’)

// All the heights given here are the total videoheight.
// Later, When we create, or Update (Crop, prescale ),
// We have to use the field heights ( 1/2 of the videoheight )

// if you are going to change these heights make sure, they
// are even numbers.

#define CROP_TOP 24
#define VIDEO_HEIGHT (CROP_TOP + CROP_HEIGHT )
#define VIDEO_WIDTH 720
#define CROP_HEIGHT 480
#define CROP_WIDTH 720
#define CROP_BOTTOM (CROP_TOP + CROP_HEIGHT)

typedef struct _VPINFO
{
    DDVIDEOPORTDESC ddVPDesc;
    DDVIDEOPORTINFO ddVPInfo;
    LPDIRECTDRAWVIDEOPORT lpVideoPort;
    int iCnt;
    LPDIRECTDRAWSURFACE lpVideoSurface;
    LPDIRECTDRAWSURFACE lpVBISurface;
} VPINFO;

//*********************************************************************
// CVpetestApp:
//*********************************************************************

class CVpetestApp
{
public:
// contructor
    CVpetestApp();
// destructor
    ~CVpetestApp();

    BOOL bOverlayVisible ;
    BOOL bVideoOn ;
```

```
// initializations
    HRESULT Initialize(HWND);
    HRESULT CreateDriver();

// surface manipulations
    HRESULT CreateSurface();
    HRESULT CreateOverlay();
    HRESULT ShowOverlay();
    HRESULT HideOverlay();

// display manipulations
    HRESULT SetColorkey() ;
    HRESULT SetDisplayMode();
    HRESULT RestoreDisplayMode();

// frame display
    void SetBobMode();

// deallocations
    void DestroyDriver();
    void ReleaseOverlay();
    void ReleasePrimarySurface();
    void ReleaseVideoPort();

// setup
    void PreSetupVideoPort();
    void SetupVideoPortforDVD() ;
    void ResetVideoPortFlags() ;

// video port information
    LPDDVIDEOPORTDESC GetddVPDesc();
    LPDDVIDEOPORTINFO GetddVPInfo();

// video port manipulations
    HRESULT CreateVideoPort(void);
    HRESULT StartVideoPort(void);
    HRESULT UpdateVideoPort(void);
    HRESULT StopVideoPort(void);

// video display information
    void GetVideoDisplayValues(RECT *rVPEsrc, SIZE *sPreScale, SIZE
    *sOverlay, SIZE *sClientWindow ) ;
    void GetVideoDisplayValues(RECT *rVPEsrc ) ;

    SIZE sVideoPortSize ;

    private:
    LPDIRECTDRAW lpDD;// The directdraw object
    GUID *lpDriverGUID;// Pointer to a unique GUID which
// represents the display device. Will be
// set to NULL,to use the default driver.
    LPDIRECTDRAWSURFACE lpdds;
    LPDIRECTDRAWSURFACE lpOverlay;
    LPDDVIDEOPORTCONTAINER lpDVP;
    VPINFO ddVideoPort;
```

```
    HWND hWndMain;

    DDPIXELFORMAT ddpfInputFormat;

    DDPIXELFORMAT ddpfVBIOutputFormat;
    DDPIXELFORMAT ddpfVBIInputFormat;
    SIZE sDisplayOverlaySize;
    BOOL bScaleAndZoom;
};

//**********************************************************************


VPE.CPP File Listing
// File Name: vpe.cpp

#include “vpe.h”
#include <stdio.h>
#include <math.h>
#include <CONIO.H>
#include “ddutil.h”

#define YUY2_4CC mmioFOURCC(‘Y’,’U’,’Y’,’2’)

BOOL fDoBob = TRUE;

extern BOOL bFull;

int iAdjust_Bottom;
int iAdjust_Right;

//**********************************************************************
// CVpetestApp
//
// CVpetestApp construction
//**********************************************************************

CVpetestApp::CVpetestApp()
{
// no specific needs
} /* CVpetestApp */

//**********************************************************************
// ~CVpetestApp
//
// CVpetestApp destruction
//**********************************************************************

CVpetestApp::~CVpetestApp()
{
    DestroyDriver();
} /* ~CVpetestApp */



//**********************************************************************
```

```
// Initialize
//
// Set the DirectDraw objects to NULL, initialize the flags, and begin // setting
up the DirectDraw objects
//*************************************************************

HRESULT CVpetestApp::Initialize(HWND hwnd)
{
    HRESULT ddrval;

    lpDD = NULL;
    lpDVP = NULL;
    lpDriverGUID = NULL; // Set to NULL to use the default disp drvr
    hWndMain = hwnd;
    bOverlayVisible = FALSE;
    bVideoOn = FALSE;
    ddrval = CreateDriver();

    return ddrval;
} /* Initialize */

//*************************************************************
// CreateDriver
//
// Create the DirectDraw object and get the video port interface
//*************************************************************

HRESULT CVpetestApp::CreateDriver()
{
    HRESULT ddrval;
    LPDIRECTDRAW lpdd;

// Create DirectDraw object, using default display adapter
    ddrval = DirectDrawCreate(lpDriverGUID, &lpdd, NULL);
    if (DD_OK == ddrval)
        lpDD = lpdd;
    else
        return ddrval;
    if (NULL == lpDVP)
    {
        // Retrieve Video Port Container Interface
        ddrval = lpDD->QueryInterface( IID_IDDVideoPortContainer,
        (LPVOID *)&lpDVP);
        if ( NULL == lpDVP )
            return ddrval;
    }
// Set application as a standard windows application
    ddrval = lpDD->SetCooperativeLevel(hWndMain, DDSCL_NORMAL);

    return ddrval;
} /* CreateDriver */


//*************************************************************
// SetDisplayMode
//
```

```
// Setup for fullscreen mode
//
//*************************************************************

HRESULT CVpetestApp::SetDisplayMode()
{
    HRESULT ddrval ;

// set flag to indicate fullscreen mode
    bFull = TRUE;

// Set control level to exclusive, fullscreen mode
    lpDD->SetCooperativeLevel(hWndMain,
    DDSCL_EXCLUSIVE|DDSCL_FULLSCREEN);

// Set the display mode to 720x480
    ddrval = lpDD->SetDisplayMode(720, 480, 0 );
    lpDD->SetCooperativeLevel(hWndMain, DDSCL_FULLSCREEN);

// clean-up routines
    ReleaseVideoPort();
    ReleaseOverlay();
    ReleasePrimarySurface();

// setup routines
    PreSetupVideoPort();
    SetupVideoPortforDVD();

// build resources
    CreateSurface();
    CreateVideoPort();
    CreateOverlay();
    SetColorkey();

// display data
    StartVideoPort();
    ShowOverlay();

    return ddrval;
} /* SetDisplayMode */

//*************************************************************
// RestoreDisplayMode
//
// Return display mode to previous setting and reset cooperative level
//*************************************************************

HRESULT CVpetestApp::RestoreDisplayMode()
{
    HRESULT ddrval;

// returns display mode to previous setting
    ddrval = lpDD->RestoreDisplayMode();

// we are currently in exclusive mode, so return to normal
    lpDD->SetCooperativeLevel(hWndMain, DDSCL_NORMAL);
```

```
    ReleaseVideoPort();
    ReleaseOverlay();
    ReleasePrimarySurface();

// set flag to indicate we are no longer in fullscreen mode
    bFull = FALSE;

    return ddrval;
} /* RestoreDisplayMode */

//*********************************************************************
// ReleaseOverlay
//
// Destroy overlay
//*********************************************************************

void CVpetestApp::ReleaseOverlay()
{
    HRESULT ddrval ;

    if ( lpOverlay != NULL )
    {
    // decrement lpOverlay's reference count (COM -- IUnknown)
        ddrval = lpOverlay->Release();
        lpOverlay = NULL ;
    }
} /* ReleaseOverlay */

//*********************************************************************
// ReleasePrimarySurface
//
// Destroy the primary surface
//*********************************************************************

void CVpetestApp::ReleasePrimarySurface()
{
    HRESULT ddrval ;

// Release DirectDraw surfaces
    if ( lpdds != NULL )
    {
// decrement lpdds' reference count (COM -- IUnknown)
        ddrval = lpdds->Release() ;
        lpdds = NULL ;
    }
} /* ReleasePrimarySurface */


//*********************************************************************
// ReleaseVideoPort
//
// Destroy the video port
//*********************************************************************

void CVpetestApp::ReleaseVideoPort()
```

```
{
    HRESULT ddrval ;
    if ( ddVideoPort.lpVideoPort != NULL )
    {
    // decrement reference count (COM -- IUnknown)
        ddVideoPort.lpVideoPort->Release();
        ddVideoPort.lpVideoPort = NULL;
    }

    //Release VideoPort
    if ( lpDVP != NULL )
    {
        ddrval = lpDVP->Release();
        lpDVP = NULL;
    }
} /* ReleaseVideoPort */

//********************************************************************
// DestroyDriver
//
// Destroy all associated objects and DirectDraw object
//********************************************************************

void CVpetestApp::DestroyDriver()
{
/* Clean up and dump all objects */
    ReleaseVideoPort();
    ReleaseOverlay();
    ReleasePrimarySurface();

//Release DirectDrawObject
    if (lpDD != NULL )
    {
        lpDD->Release();
        lpDD = NULL;
    }
} /* DestroyDriver */

//********************************************************************
// GetddVPDesc
//
// Access to the video port description
//********************************************************************

LPDDVIDEOPORTDESC CVpetestApp::GetddVPDesc()
{
    LPDDVIDEOPORTDESC lpddVPDesc;

    lpddVPDesc = &(ddVideoPort.ddVPDesc);

    return lpddVPDesc;
} /* GetddVPDesc */

//********************************************************************
// GetddVPInfo
//
```

```
// Access to video port information
//*************************************************************

LPDDVIDEOPORTINFO CVpetestApp::GetddVPInfo()
{
    LPDDVIDEOPORTINFO lpddVPInfo;

    lpddVPInfo = &(ddVideoPort.ddVPInfo);

    return lpddVPInfo;
} /* GetddVPInfo */

//*************************************************************
// PreSetupVideoPort
//
// Add all video port standard fare with VBI and YUV
//*************************************************************

void CVpetestApp::PreSetupVideoPort()
{
    LPDDVIDEOPORTDESC lpddVPDesc;
    LPDDVIDEOPORTINFO lpddVPInfo;

    lpddVPDesc = &(ddVideoPort.ddVPDesc);
    lpddVPInfo = &(ddVideoPort.ddVPInfo);

// init videoport description
    memset(lpddVPDesc, 0, sizeof(DDVIDEOPORTDESC)); // block memory
    lpddVPDesc->dwSize = sizeof(DDVIDEOPORTDESC);
    lpddVPDesc->dwMicrosecondsPerField = 16000; // Any Non-0 value;
    lpddVPDesc->dwMaxPixelsPerSecond = 8000; // Any Non Zero Value;
    lpddVPDesc->dwVideoPortID = 0; // Should be Zero

// init videoport connect info
    lpddVPDesc->VideoPortType.dwSize = sizeof(DDVIDEOPORTCONNECT);
    memcpy(&(lpddVPDesc->VideoPortType.guidTypeID),
    &DDVPTYPE_E_HREFL_VREFL, sizeof(GUID));

// init videoport info
    memset(lpddVPInfo, 0, sizeof(DDVIDEOPORTINFO));
    lpddVPInfo->dwSize = sizeof(DDVIDEOPORTINFO);
    lpddVPInfo->dwOriginX = 0;
    lpddVPInfo->dwOriginY = 0;
    lpddVPInfo->dwVBIHeight = 0 ;
    lpddVPInfo->lpddpfInputFormat = &ddpfInputFormat;
// format written to video port


// Output format of the VBI data
    lpddVPInfo->lpddpfVBIOutputFormat = &ddpfVBIOutputFormat;
// Input format of the VBI data
    lpddVPInfo->lpddpfVBIInputFormat = &ddpfVBIInputFormat;

    memset(&ddpfInputFormat, 0, sizeof(DDPIXELFORMAT));
    memset(&ddpfVBIInputFormat, 0, sizeof(DDPIXELFORMAT));
    memset(&ddpfVBIOutputFormat, 0, sizeof(DDPIXELFORMAT));
```

```
    ddpfInputFormat.dwFlags = DDPF_FOURCC; // Using YUV surfaces
    ddpfInputFormat.dwFourCC = YUY2_4CC;

    lpddVPInfo->lpddpfInputFormat->dwSize = sizeof(DDPIXELFORMAT);
    lpddVPInfo->lpddpfVBIInputFormat->dwSize = sizeof(DDPIXELFORMAT);
    lpddVPInfo->lpddpfVBIOutputFormat->dwSize= sizeof(DDPIXELFORMAT);

    ddpfVBIInputFormat.dwFlags = DDPF_FOURCC; // Using YUV surfaces
    ddpfVBIInputFormat.dwFourCC = YUY2_4CC;
} /* PreSetupVideoPort */

//*************************************************************
// SetupVideoPortDVD
//
// Set up the video port for default 740x480, 8 bit port configuration
//*************************************************************

void CVpetestApp::SetupVideoPortforDVD()
{
    PreSetupVideoPort();

    LPDDVIDEOPORTDESC lpddVPDesc;
    LPDDVIDEOPORTINFO lpddVPInfo;

    lpddVPDesc = &(ddVideoPort.ddVPDesc);
    lpddVPInfo = &(ddVideoPort.ddVPInfo);
    iAdjust_Bottom = 2;// size adjustments
    iAdjust_Right = 8;
    lpddVPInfo->dwVPFlags = 0;
    sVideoPortSize.cx = VIDEO_WIDTH + iAdjust_Right;
    sVideoPortSize.cy = VIDEO_HEIGHT + iAdjust_Bottom;
    lpddVPInfo->dwOriginX = 0;
    lpddVPInfo->dwOriginY = 0;

    // fields are 1/2 size
    iAdjust_Bottom /= 2;
    lpddVPDesc->dwFieldHeight = sVideoPortSize.cy / 2;

    if (fDoBob)
    {
        lpddVPDesc->VideoPortType.dwFlags = 0; // MUST use
    // Noninterlaced
    }
    else
    {
        lpddVPDesc->VideoPortType.dwFlags = DDVPCONNECT_INTERLACED;
        lpddVPInfo->dwVPFlags |= DDVP_INTERLEAVE;
    }


    // field width is the same
    lpddVPDesc->dwFieldWidth = sVideoPortSize.cx + iAdjust_Right;
    lpddVPDesc->dwVBIWidth = lpddVPDesc->dwFieldWidth;
    lpddVPDesc->VideoPortType.dwPortWidth = 8;
    lpddVPDesc->VideoPortType.dwFlags |= DDVPCONNECT_VACT;
```

```
    lpddVPDesc->VideoPortType.dwFlags |= DDVPCONNECT_DOUBLECLOCK;
    lpddVPInfo->dwVPFlags |= DDVP_AUTOFLIP;
    lpddVPInfo->rCrop.left = 0;

    // Microsoft requires the rCrop.right to be inclusive of endpoint
    lpddVPInfo->rCrop.right = sVideoPortSize.cx; //640 ;

    // We are dividing by 2, so that the value will be in FieldHeight
    lpddVPInfo->rCrop.top = CROP_TOP / 2; //20 ;

    // Microsoft requires the rCrop.bottom be inclusive of endpoint
    lpddVPInfo->rCrop.bottom = lpddVPDesc->dwFieldHeight;
    lpddVPInfo->dwVPFlags |= DDVP_CROP;
} /* SetupVideoPortforDVD */

//*********************************************************************
// ResetVideoPortFlags
//
// Set video port flags to default values
//*********************************************************************

void CVpetestApp::ResetVideoPortFlags()
{
    ddVideoPort.ddVPInfo.dwVPFlags = DDVP_AUTOFLIP;
    if (fDoBob)
    {
        ddVideoPort.ddVPDesc.VideoPortType.dwFlags =
        DDVPCONNECT_VACT | DDVPCONNECT_DOUBLECLOCK;
    }
    else
    {
        ddVideoPort.ddVPDesc.VideoPortType.dwFlags =
        DDVPCONNECT_VACT | DDVPCONNECT_INTERLACED |
        DDVPCONNECT_DOUBLECLOCK;
        ddVideoPort.ddVPInfo.dwVPFlags |= DDVP_INTERLEAVE;
    }
} /* ResetVideoPortFlags */

//*********************************************************************
// CreateVideoPort
//
// Obtain the video port interface and create the port
//*********************************************************************

HRESULT CVpetestApp::CreateVideoPort()
{
    HRESULT ddRVal;


    if ( NULL == lpDVP) // check the video port container
    {
    // Retrieve Video Port Container Interface
        ddRVal = lpDD->QueryInterface( IID_IDDVideoPortContainer,
        (LPVOID *)&lpDVP);
    if ( NULL == lpDVP )
        return ddRVal;
```

```
    }

    if ( ddVideoPort.lpVideoPort != NULL ) // container is not empty
        ddVideoPort.lpVideoPort->Release(); // release container

    // create the video port
    ddRVal = lpDVP->CreateVideoPort(0L, &(ddVideoPort.ddVPDesc),
    &(ddVideoPort.lpVideoPort), NULL);

    return ddRVal;
} /* CreateVideoPort */

//*********************************************************************
// CreateSurface
//
// Creates a primary direct draw surface.
//*********************************************************************

HRESULT CVpetestApp::CreateSurface()
{
    HRESULT ddrval;
    DDSURFACEDESC ddsd;

    // setup surface info
    ddsd.dwSize = sizeof ( ddsd );
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

    // create the surface
    ddrval = lpDD->CreateSurface( &ddsd, &lpdds, NULL );

    return ddrval;
} /* CreateSurface */

//*********************************************************************
// CreateOverlay
//
// Create a DirectDraw surface as an overlay
//*********************************************************************

HRESULT CVpetestApp::CreateOverlay()
{
    HRESULT ddrval;
    DDSURFACEDESC ddsd;

    ddsd.dwSize = sizeof ( ddsd );
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.dwFlags |= DDSD_HEIGHT | DDSD_WIDTH; // set height, width
    ddsd.dwWidth = sVideoPortSize.cx;
    ddsd.dwHeight = sVideoPortSize.cy;
    ddsd.dwFlags |= DDSD_PIXELFORMAT; // set pixel format
    ddsd.ddpfPixelFormat.dwSize = 0 ;
    ddsd.ddpfPixelFormat.dwFlags = DDPF_FOURCC; // using YUV format
    ddsd.ddpfPixelFormat.dwFourCC = YUY2_4CC;
    ddsd.ddpfPixelFormat.dwRGBBitCount = 16; // using 16-bit color
    ddsd.dwFlags |= DDSD_BACKBUFFERCOUNT; // adding one back buffer
```

```
    ddsd.dwBackBufferCount = 1;
    ddsd.ddsCaps.dwCaps = DDSCAPS_COMPLEX; // overlay surface
    //characteristics
    ddsd.ddsCaps.dwCaps |= DDSCAPS_OVERLAY;
    ddsd.ddsCaps.dwCaps |= DDSCAPS_FLIP;
    ddsd.ddsCaps.dwCaps |= DDSCAPS_VIDEOPORT;
    ddrval = lpDD->CreateSurface( &ddsd, &lpOverlay, NULL );

    return ddrval;
} /* CreateOverlay */

//*********************************************************************
// ShowOverlay
//
// Display an overlay on the screen
//*********************************************************************

HRESULT CVpetestApp::ShowOverlay()
{
    LPDIRECTDRAWSURFACE psurf;
    LPDIRECTDRAWSURFACE pdest;
    HRESULT ddrval = DD_OK;
    RECT srcrect;
    RECT destrect;
    RECT windowrect;
    RECT clientrect;
    char debugstring[80];
    DDOVERLAYFX dofx;

    // check to see if overlay and surface are lost
    // (in case another application required the memory)
    if (lpOverlay->IsLost())
        lpOverlay->Restore(); // restore the overlay memory

    if (lpdds->IsLost())
        lpdds->Restore(); // restore the surface memory

    GetWindowRect( hWndMain, &windowrect );
    GetClientRect( hWndMain, &clientrect );
    if ( bVideoOn )
    {
        if (( clientrect.right - clientrect.left < 1 ) ||
        ( clientrect.bottom - clientrect.top < 1 ))
        {
        // turn the overlay off
            ddrval = HideOverlay();
        }
        else
        {

        // Obtain the video width and height
            int iVideoWidth = ddVideoPort.ddVPDesc.dwFieldWidth;
            int iVideoHeight = ddVideoPort.ddVPDesc.dwFieldHeight;

            if ( ddVideoPort.ddVPInfo.dwVPFlags & DDVP_CROP )
            {
```

```
// Get cropped size information
    iVideoWidth = ddVideoPort.ddVPInfo.rCrop.right -
    ddVideoPort.ddVPInfo.rCrop.left;
    iVideoHeight = ddVideoPort.ddVPInfo.rCrop.bottom -
    ddVideoPort.ddVPInfo.rCrop.top;
}

// Set the video source rectangle(area in the overlay
// surface)
srcrect.left = 0;
srcrect.top = 0;

// The source rectangle is defined by x,y & height & width
srcrect.right = iVideoWidth - iAdjust_Right;
srcrect.bottom = iVideoHeight - iAdjust_Bottom;

// Set the destination rectangle area in primary surface
int iWidth = clientrect.right - clientrect.left;
// divide by two to convert to FieldHeight
int iHeight = ( clientrect.bottom - clientrect.top ) / 2;

// Obtain the prescale factors
if ( (iWidth + iAdjust_Right < iVideoWidth ) || (iHeight +
iAdjust_Bottom < iVideoHeight ) )
{
    // dest rectangle is smaller than the src rectangle
    LONG tempX = iWidth + iAdjust_Right;
    LONG tempY = iVideoHeight;

    // the i740 scalar requires the width to be a
    // multiple of 16.
    tempX = tempX - tempX % 16;
    // halve the Y value until it is less than dest
    // height
    while ( tempY > iHeight + iAdjust_Bottom )
        tempY /= 2;

    ddVideoPort.ddVPInfo.dwPrescaleWidth = tempX;
    ddVideoPort.ddVPInfo.dwPrescaleHeight = tempY;
    srcrect.right = ddVideoPort.ddVPInfo.dwPrescaleWidth
    - iAdjust_Right;
    srcrect.bottom =ddVideoPort.ddVPInfo.dwPrescaleHeight
    - iAdjust_Bottom;

    // set video port flags to prescale
    DWORD dwTempFlags;
    dwTempFlags = ddVideoPort.ddVPInfo.dwVPFlags;
    ddVideoPort.ddVPInfo.dwVPFlags |= DDVP_PRESCALE;

    // update the video
    ddrval = UpdateVideoPort();


    if ( ddrval != DD_OK )
    {
        sprintf(debugstring, “Error occurred at %d %d
```

```
            %d %d\n",
            ddVideoPort.ddVPInfo.dwPrescaleWidth,
            ddVideoPort.ddVPInfo.dwPrescaleHeight,
            srcrect.right, srcrect.bottom);
            MessageBox(hWndMain,debugstring,"VPE
            Error",MB_OK);

            return ddrval;
        }
    }
    else
    {
        // do not prescale
        ddVideoPort.ddVPInfo.dwVPFlags &= ~DDVP_PRESCALE;

        // update the video
        ddrval = UpdateVideoPort();
        if ( ddrval != DD_OK )
            return ddrval;
    }
    if ( fDoBob == FALSE)
        srcrect.bottom *= 2 ; // bob algorithm uses 1/2 ht

    if ( srcrect.right > OVERLAY_MAX_WIDTH )
    // make sure the overlay isn't too wide
        srcrect.right = OVERLAY_MAX_WIDTH - 1;

    if ( srcrect.bottom > OVERLAY_MAX_WIDTH )
     // make sure the overlay isn't too high
        srcrect.bottom = OVERLAY_MAX_HEIGHT - 1;

        POINT lefttop, rightbottom;
        lefttop.x = clientrect.left;
        lefttop.y = clientrect.top;
        rightbottom.x = clientrect.right;
        rightbottom.y = clientrect.bottom;

        // convert client coordinates to screen coordinates
        ClientToScreen( hWndMain, &lefttop );
        ClientToScreen( hWndMain, &rightbottom);
        destrect.left = lefttop.x;
        destrect.right = rightbottom.x;
        destrect.top = lefttop.y;
        destrect.bottom = rightbottom.y;

        // get surface ptrs
        psurf = lpOverlay;
        pdest = lpdds;

        // set up overlay fx
        memset( &dofx, 0, sizeof( dofx ) );
        dofx.dwSize = sizeof( dofx );


        // crop when off the edge of the screen
        float fZoomX = ( (float)( destrect.right - destrect.left )
```

```
/ ( srcrect.right - srcrect.left ) );
float fZoomY = ( (float)( destrect.bottom - destrect.top )
/ ( srcrect.bottom - srcrect.top ) );

// get screen dimensions (resolution)
int screenX = GetSystemMetrics(SM_CXSCREEN);
int screenY = GetSystemMetrics(SM_CYSCREEN);

// check the dest rectangle size and modify if needed
if (destrect.bottom > screenY )
{
    srcrect.bottom -= (int) (( destrect.bottom -
    (screenY - 1) ) / fZoomY );
    destrect.bottom = screenY;
}

if (destrect.right > screenX )
{
    srcrect.right -= (int) ( ( destrect.right -
    (screenX - 1) ) / fZoomX );
    destrect.right = screenX;
}

if (destrect.top < 0)
{
    srcrect.top = (int)( ( 0 - destrect.top ) / fZoomY );
    destrect.top = 0;
}
else
    srcrect.top = 0;

if (destrect.left < 0)
{
    srcrect.left = (int)(( 0 - destrect.left ) / fZoomX);
    destrect.left = 0;
}
else
    srcrect.left = 0;
// check that the new rect dimensions render it visible
if (( srcrect.right - srcrect.left <= 0 ) ||
( srcrect.bottom - srcrect.top <= 0 ))
{
// turn the overlay off
    ddrval = HideOverlay();
}
else
{
    DWORD dwFlags;

    // set flags
    if (fDoBob)
        dwFlags = DDOVER_SHOW | DDOVER_KEYDEST |
        DDOVER_AUTOFLIP | DDOVER_BOB |
        DDOVER_REFRESHDIRTYRECTS;
    else
        dwFlags = DDOVER_SHOW | DDOVER_KEYDEST |
```

```
                        DDOVER_AUTOFLIP |
                        DDOVER_REFRESHDIRTYRECTS;

                    // update the overlay
                    ddrval = psurf->UpdateOverlay(&srcrect,
                    pdest,&destrect, dwFlags, NULL);

                    if ( ddrval != DD_OK )
                    {
                        sprintf(debugstring, “Error occurred at %d %d
                        %d %d == %d %d %d %d\n”, srcrect.left,
                        srcrect.top, srcrect.right,
                        srcrect.bottom, destrect.left,
                        destrect.top, destrect.right,
                        destrect.bottom);
                        OutputDebugString(debugstring);

                        return ddrval;
                    }
                    else
                        bOverlayVisible = TRUE;
                }
                // update sDisplayOverlaySize (This is used for status
                // display)
                sDisplayOverlaySize.cx = srcrect.right - srcrect.left + 1;

                if (fDoBob)
                    sDisplayOverlaySize.cy = (srcrect.bottom -
                    srcrect.top) * 2 + 1;
                else
                    sDisplayOverlaySize.cy = srcrect.bottom - srcrect.top
                    + 1;
            }
    }

    return ddrval;
} /* ShowOverlay */

//*********************************************************************
// HideOverlay
//
// Turn the overlay off
//*********************************************************************

HRESULT CVpetestApp::HideOverlay()
{
    HRESULT ddrval;
    // turn the overlay off
    ddrval = lpOverlay ->UpdateOverlay(NULL, lpdds, NULL,
    DDOVER_HIDE | DDOVER_REFRESHDIRTYRECTS, NULL );
    if ( ddrval == DD_OK )
    bOverlayVisible = FALSE;

    return ddrval;
} /* HideOverlay */
//*********************************************************************
```

```
// SetColorKey
//
// Match the color and set it as the transparent color.
//****************************************************************

HRESULT CVpetestApp::SetColorkey()
{
    HRESULT ddrval;
    DDCOLORKEY ddck;

    ddck.dwColorSpaceLowValue = DDColorMatch(lpdds,
    RGB(0xff,0x00,0xff));
    ddck.dwColorSpaceHighValue = ddck.dwColorSpaceLowValue;

    ddrval = lpOverlay ->SetColorKey(DDCKEY_DESTOVERLAY, &ddck );
    if ( ddrval == DD_OK )
        ddrval = lpdds ->SetColorKey(DDCKEY_DESTOVERLAY, &ddck );

    return ddrval = DD_OK ;
} /* SetColorkey */

//****************************************************************
// StartVideoPort
//
// Gets the video port options (displays the dialog) and then tells
// DDRAW to start the video.
//****************************************************************

HRESULT CVpetestApp::StartVideoPort()
{
    HRESULT ddRVal;
    // designate the overlay as the recipient of the vid. data stream
    if ( ddVideoPort.lpVideoPort )
        ddRVal = ddVideoPort.lpVideoPort->
        SetTargetSurface(lpOverlay, DDVPTARGET_VIDEO);

    // start the flow of video data
    if ( ddRVal == DD_OK )
        ddRVal = ddVideoPort.lpVideoPort->
        StartVideo(&(ddVideoPort.ddVPInfo));

    if ( ddRVal == DD_OK )
        bVideoOn = TRUE;

    return ddRVal;
} /* StartVideoPort */


//****************************************************************
// UpdateVideoPort
//
// Gets the video port options (displays the dialog) and then tells
// DDRAW to update the video.
//****************************************************************

HRESULT CVpetestApp::UpdateVideoPort()
```

```
{
    HRESULT ddRVal;

    // update the video
    ddRVal = ddVideoPort.lpVideoPort->
    UpdateVideo(&(ddVideoPort.ddVPInfo));
    if ( ddRVal == DD_OK )
        bVideoOn = TRUE;

    return ddRVal;
} /* UpdateVideoPort */

//*********************************************************************
// StopVideoPort
//
// Tells DirectDraw to stop the video.
//*********************************************************************

HRESULT CVpetestApp::StopVideoPort()
{
    HRESULT ddRVal;

    ddRVal = ddVideoPort.lpVideoPort->StopVideo();
    if ( ddRVal == DD_OK )
        bVideoOn = FALSE;

    return ddRVal;
} /* StopVideoPort */

//*********************************************************************
// GetVideoDisplayValues
//
// Copy video port display values to the rectangle pointer
//*********************************************************************

void CVpetestApp::GetVideoDisplayValues(RECT *rVPESrc, SIZE *sPreScale,
SIZE *sOverlay, SIZE *sClientWindow )
{
    // if there is a cropping rectangle, then use it instead of the
    // field dimensions
    if ( ddVideoPort.ddVPInfo.dwVPFlags & DDVP_CROP )
        *rVPESrc = ddVideoPort.ddVPInfo.rCrop;
    else
    {
        rVPESrc->left = 0;
        rVPESrc->top = 0;
        rVPESrc->right = ddVideoPort.ddVPDesc.dwFieldWidth - 1;
        rVPESrc->bottom = ddVideoPort.ddVPDesc.dwFieldHeight - 1;
    }
    // if prescale values are used, use those factors
    if ( ddVideoPort.ddVPInfo.dwVPFlags & DDVP_PRESCALE )
    {
        sPreScale->cx = ddVideoPort.ddVPInfo.dwPrescaleWidth;
        sPreScale->cy = ddVideoPort.ddVPInfo.dwPrescaleHeight;
    }
    else
```

```
    {
        sPreScale->cx = -1;
        sPreScale->cy = -1;
    }

    *sOverlay = sDisplayOverlaySize;
    RECT clientrect;
    GetClientRect( hWndMain, &clientrect );

    // set width (cx) & height (cy)
    sClientWindow->cx = clientrect.right - clientrect.left + 1;
    sClientWindow->cy = clientrect.bottom - clientrect.top + 1;

} /* GetVideoDisplayValues */

//*************************************************************
// GetVideoDisplayValues
//
// Copy video port display values to the rectangle pointer
//*************************************************************

void CVpetestApp::GetVideoDisplayValues(RECT *rVPESrc )
{
    // if there is a cropping rectangle, then use it instead of
    // the field dimensions
    if ( ddVideoPort.ddVPInfo.dwVPFlags & DDVP_CROP )
        *rVPESrc = ddVideoPort.ddVPInfo.rCrop;
    else
    {
        rVPESrc->left = 0;
        rVPESrc->top = 0;
        rVPESrc->right = ddVideoPort.ddVPDesc.dwFieldWidth - 1;
        if (fDoBob)
            rVPESrc->bottom = ddVideoPort.ddVPDesc.dwFieldHeight
            * 2 - 1;
        else
            rVPESrc->bottom = ddVideoPort.ddVPDesc.dwFieldHeight
            - 1;
    }
} /* GetVideoDisplayValues */
```